ASYNCHRONOUS GENETIC CIRCUIT DESIGN

by

Tramy T Nguyen

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 Electrical and Computer Engineering

The University of Utah

December 2019

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Tramy T Nguyen**
has been approved by the following supervisory committee members:

Chris J. Myers,	Chair(s)	**August 29, 2019** Date Approved
Kenneth Stevens,	Member	**August 29, 2019** Date Approved
Priyank Kalla,	Member	**August 29, 2019** Date Approved
Tara Deans,	Member	**August 29, 2019** Date Approved
Nicholas Roehner,	Member	**September 04, 2019** Date Approved

by **Florian Solzbacher**, Chair/Dean of
the Department/College/School of **Electrical and Computer Engineering**
and by **David B. Kieda**, Dean of The Graduate School.

ABSTRACT

Synthetic biology is applying engineering concepts to biological processes to enable genetic circuit designs, among other applications. As more biological parts are being discovered, it is vital to have an automated procedure to allow complex circuit designs to be built. Technology mapping is a set of procedures that maps biological components to a design specification. Current technology mapping frameworks for genetic circuits are used to design combinational circuits. This dissertation illustrates the process of building an automated workflow for a technology mapping framework to design asynchronous sequential genetic circuits. An automated process to create a library of gates for logic and memory circuits is described to construct gates from DNA parts retrieved from a standardize data repository. Genetic constraints address what parts can be mapped to the design specification when the gates and designs are constructed. The proposed automaton workflow begins with a specification provided in a formal design language, such as Verilog. The input design specification is converted into a genetic regulatory network represented using the *Synthetic Biology Open Language* (SBOL). The network is decomposed into base functions (NOR gates, inverters, and genetic toggle switches) and matching and covering algorithms are performed to produce the output design. The output design is converted to the *Systems Biology Markup Language* (SBML) data format for testing and simulation. The outcome of this work provides the synthetic biology community insights on how asynchronous sequential circuit designs can be built through an automated procedure to perform technology mapping from libraries composed of logic gates and memory circuits.

For my parents.

CONTENTS

ABSTRACT . . . . . **iii**

LIST OF FIGURES . . . . . **vii**

LIST OF TABLES . . . . . **xvii**

LIST OF ALGORITHMS . . . . . **xviii**

ACKNOWLEDGEMENTS . . . . . **xix**

CHAPTERS

1. INTRODUCTION **1**

1.1 Genetic Design Automation (GDA) 2
1.2 Contributions 6
1.3 Dissertation Outline 8

2. BACKGROUND **10**

2.1 Asynchronous Circuit Design 10
2.2 Genetic Circuits 14
2.3 Biological Data Standards 17
2.3.1 Synthetic Biology Open Language 18
2.3.2 Systems Biology Markup Language 20
2.4 iBioSim 21

3. ASYNCHRONOUS GENETIC CIRCUIT DESIGN **24**

3.1 Specification 25
3.2 Synthesis 29
3.2.1 Behavioral Verilog to SBML 32
3.2.2 SBML to LPN 33
3.2.3 Synthesizing in ATACS 33
3.2.4 Structural Verilog to SBOL 35
3.3 Summary 36

4. PROGRAMMATIC CREATION OF GATE LIBRARIES THROUGH AUTOMATED GATE GENERATION **38**

4.1 Assembling Transcriptional Units 40
4.2 Gate Generation 42
4.3 Identifying Gate Types 43
4.4 Summary 48

5. TECHNOLOGY MAPPING FOR ASYNCHRONOUS GENETIC CIRCUITS . . . 50

5.1 Boolean Decomposition . . . 51
5.2 Matching . . . 54
5.3 Covering . . . 59
5.3.1 Exhaustive Covering Algorithm . . . 60
5.3.2 Greedy Covering Algorithm . . . 68
5.3.3 Branch and Bound Covering . . . 68
5.4 Generating a Netlist of Genetic Gates Encoded into SBOL . . . 75
5.5 Summary . . . 78

6. VERIFICATION . . . 80

6.1 Synthesis Verification . . . 81
6.2 Genetic Circuit Verification . . . 83
6.3 Summary . . . 84

7. CASE STUDY: GENETIC LOW-PASS FILTER . . . 85

7.1 Cello Library Gates . . . 85
7.2 Genetic Low-Pass Filter . . . 86
7.3 Results of Our Technology Mapping Procedure . . . 90
7.4 Discussion . . . 93

8. CONCLUSION . . . 96

8.1 Summary . . . 96
8.2 Future Work . . . 98
8.2.1 ATACS Search Space . . . 98
8.2.2 Replicating ATACS for Genetic Circuits . . . 98
8.2.3 Curate Part Library into SBOL . . . 98
8.2.4 Evaluate Promoter Location to Assemble Transcriptional Units . . . 99
8.2.5 Support Additional Genetic Gate Types . . . 99
8.2.6 Expand Signal Mismatch . . . 100
8.2.7 Evaluating Threshold of Genetic Gates Before Cell Death . . . 100
8.2.8 Increase Technology Mapping Performance . . . 100
8.2.9 Reduce Structural Bias in Technology Mapping . . . 101
8.2.10 Enriched Parts to Improve Genetic Circuit Models . . . 101

REFERENCES . . . 102

LIST OF FIGURES

1.1 A high-level overview of an EDA workflow to build an electronic circuit. First, the behavior of a circuit is described in a high-level descriptive language. The behavioral design is then synthesized to a structural design that describes a gate-level description. Technology mapping is performed to produce a layout of what electronic components are needed to physically build the circuit. Verification is performed to ensure the layout of the physical design meets the requirement of the specification described in the behavioral design. 2

1.2 This figure shows a high-level GDA workflow. This GDA workflow is similar to the EDA workflow except the physical design generates DNA sequences. . . 3

1.3 This workflow is illustrated using a genetic toggle switch design. The workflow starts off with an asynchronous design specified using behavioral Verilog. Next, the ATACS asynchronous design tool synthesizes logic equations represented using structural Verilog. The synthesized circuit is then realized as a physical design by a technology mapping procedure that selects gates from a SBOL encoded gate library stored in a `SynBioHub` repository. In order to verify the workflow, a model is generated and simulations are performed to verify that the design displays the expected behavior. 7

2.1 This figure shows the difference between an asynchronous and a synchronous S-R latch and a waveform illustrating the behavior of both circuits after they have been initialized for simulation. The S-R latch is a level-sensitive circuit designed asynchronously for storing information when there is a change on the input signals. The clocked S-R latch is a level-sensitive circuit designed synchronously for storing information based on a clock level. Set, S, and Reset, R, are input signals used for setting the circuit's data value. $\bar{Q}_{\mathrm{A}}$ is the inverse signal of Q_{A} and are both represented as output signals to show what data value is currently stored within the S-R latch. The same can be said for Q_{S} and $\bar{Q}_{\mathrm{S}}$. $\bar{Q}_{\mathrm{A}}$ and $\bar{Q}_{\mathrm{S}}$ are simplified in the waveform as the LOW state of Q_{A} and Q_{S}. The behavior of the clocked S-R latch is identical to that of the S-R latch except that the output signal is governed by a clock, CLK, signal. In other words, the output signals for Q_{S} and $\bar{Q}_{\mathrm{S}}$ are designed in this circuit to update when CLK is HIGH. The red signals that go HIGH on S is a glitch that appeared for a short duration and causes Q_{A} to go HIGH unexpectedly. The clocked S-R latch is able to avoid this glitch because the input signal was not stable before CLK could capture this information. The S-R latch, however, encounters the glitch as a change in the input signal and produced an invalid output signal. 12

2.2 A *NOT* gate and a *NOR* gate represented as a genetic circuit. The symbol of the logic gate is shown on the left. A listing of all possible combinations of the gate's input signals (X, X_0, and X_1) mapped to the output signal, Y, is shown in the Boolean table at the center. A design of a genetic circuit that behaves similarly to that of its logic gate are shown on the right. These genetic circuits are made of transcriptional units that undergoes the process of transcription and translation to transform DNA to protein. A transcriptional unit is composed of promoters, *Pro*, ribosome binding sites, RBS, coding sequences CDS, and terminators, *Ter*. The encoded protein translated from CDS is illustrated in both genetic circuits as protein Y and are represented as the output protein for the circuit. The production of the output protein Y can be controlled by proteins that can bind to the the transcriptional unit through repression or activation of the promoter. These proteins are referred to as transcription factors acting as input proteins to a genetic circuit and are represented in this example as X, X_0, and X_1. X, repressing *Pro* in the *NOT* gate example, prevents the production of the Y protein. If X is removed, this allows for protein Y to be produced. The same concept applies for building a genetic circuit that behaves similarly to a *NOR* gate. The presence of X_0 and X_1 represses *Pro*. As long as one input protein is present, the output protein Y cannot be produced. If both input proteins are removed, then protein Y is produced. 15

2.3 A genetic toggle switch is a genetic circuit that behaves similarly to that of an S-R latch. The genetic toggle switch shown in this example is composed of a transcriptional unit that primarily produces a LacI protein and a TetR protein. The *green fluorescent protein* (GFP) serves as an output reporter to indicate when the TetR protein is produced. Both the LacI protein and the TetR protein can mutually repress each other through the negative feedback loops that affect the production of its counter protein. This means that, when a LacI protein is produced, the pLac promoter is repressed, thus preventing the production of a TetR protein. Likewise, if a TetR protein is produced, the pTet promoter is repressed and no LacI protein is produced. The LacI protein and the TetR protein can be controlled by IPTG and aTc to switch between states of the two proteins. IPTG and aTc are small molecules that act as input sensors for controlling the production of the LacI and TetR protein. When IPTG is applied, it forms a complex with the LacI protein and acts as an inhibitor to the LacI protein. If the LacI protein is repressed, the pLac promoter is free to synthesize the TetR protein. Similarly, when aTc is applied, it forms a complex with the TetR protein and acts as an inhibitor to the TetR protein. If the TetR protein is repressed, the pTet promoter is free to synthesize the LacI protein. If both input sensors for IPTG and aTc are removed, then the genetic toggle switch produces whatever output protein that was synthesized in its previous state. 17

2.4 A genetic toggle switch modeled hierarchically using SBOL visual symbols to represent the SBOL data model. The circuit and its inverters are described using SBOL *ModuleDefinitions*. Instantiation of the LacI Inverter and TetR Inverter as referred to in SBOL as *Modules*. These *Modules* are connected through the use of SBOL *MapsTo*. The DNA, small molecules, proteins, and their complexes that are shown in this example are described in SBOL as *ComponentDefinitions*. Instantiations of these *ComponentDefinition* within a *ModuleDefinition* are referred to as *FunctionalComponents*. *ComponentDefinitions* can also be instantiated onto other *ComponentDefinitions* as SBOL *Components* to construct a transcriptional unit that is shown in the LacI Inverter and TetR Inverter. SBOL *Interactions* are used to specify the relationship between *FunctionalComponents* such as the forming of a complex between aTc and TetR and IPTG and LacI. SBOL *Interactions* are used in the LacI Inverter and TetR Inverter to represent the inhibition of TetR repressing pTet and LacI repressing pLac. SBOL *Interactions* are also used in both inverters to indicate the production of the LacI protein, TetR protein, and GFP protein. 18

2.5 A genetic toggle switch modeled hierarchically and is visualized using SBGN graphical symbols to describe the SBML data model. The Genetic Toggle Switch is represented as an SBML *Model*. The LacI Inverter and TetR Inverter are created as SBML *ModelDefinitions* and are instantiated inside of the Genetic Toggle Switch as *SubModels*. *Replacement* and *ReplacedBy* are used for connecting the elements within the Genetic Toggle Switch and the inverters. Small molecules, proteins, complexes, and nucleic acid that are shown in this example are represented as SBML *Species*. The forming of a complex between aTc and TetR and IPTG and LacI is described as SBML *Reactions*. In these type of *Reactions*, the complexes are referred to as the *Products* and the small molecules and the protein used to form the complex are referred to as *Modifiers*. SBML *Reactions* are also used to describe the LacI protein, TetR protein, and GFP protein produced when TetR protein is not inhibiting the pTet promoter and the LacI protein is not repressing the pLac promoter. The TetR protein repressing the pTet promoter and the LacI protein repressing the pLac promoter are referred to as *Modifiers* in their *Reactions*. The LacI protein, TetR protein, and GFP proteins produced in their *Reactions* are represented as *Products*. 21

2.6 This is a high-level diagram of the genetic circuit design workflow supported by iBioSim. The red arrows indicate the flow between the different software components and dotted lines indicate the output of each step that is then used by the proceeding software component in the workflow. First, genetic parts encoded using SBOL are fetched from SynBioHub using the SBOLDesigner plugin to construct the DNA-level design encoded using SBOL. Next, the DNA design is augmented with interaction data using the Virtual Parts model generator, and the functional SBOL is converted into an SBML model. The resulting mathematical model can then be refined and parameters configured using iBioSim's model editor. The SBML model can be analyzed in iBioSim as described by an associated SED-ML document. The data created for the SBOL parts, the SBML model, and the analysis can be shared and documented by uploading these artifacts to SynBioHub as a COMBINE archive. 22

3.1 The data workflow demonstrates how a behavioral Verilog is synthesized into a structural Verilog. ATACS is an asynchronous synthesis tool used in this workflow for transforming a high-level behavioral design into a hazard-free structural design. The structural Verilog is transformed into SBOL at the end so that physical biological parts can be realized onto the structural design. A mapping of the four data formats supported in the proposed workflow is shown under Data Conversion. The green label represents the terminologies that are supported from SBOL data model, the orange labels represent the Verilog constructs that are used for designing an asynchronous genetic circuit, the blue labels represent the elements that are supported from the SBML data model, and the white label with black boarders represents terminologies that are supported within ATACS for building an LPN model. 26

3.2 Specification of an S-R latch expressed in the Verilog language. This design style follows that of a Mealy machine in which the output signal is calculated base on the current inputs and the current state that was sensed. The *InitialBlock* is executed at the beginning to set the initial state and the output signals of the circuit. The *AlwaysBlock* is executed afterwards by first waiting on the desired input signals to become true before the circuit kickstarts and performs operations to produce an output signal. When the input signal matches what the circuit expects, then the output signal q is updated base on what input signals were sensed. The state of the circuit is updated after setting the output signal to reflect what the current output signal is set to in response to the input signals that were detected. 27

3.3 Testbench of an S-R latch expressed in Verilog. Variations of input signals are set to test the assignment of the output signal q produced from the specification. The output signal is asserted through *WaitStatements*. If the output signal q produces in invalid value, then the *WaitStatements* is locked in that state and further input signal variations are prevented from changing. 28

3.4 An example of the SR latch LPN models generated from ATACS. The left LPN model represents the specification transformed from **Figure 3.2** and the right LPN model represents the testbench transformed from **Figure 3.3**. The circles are *Places* to represent conditions. The black filled circle is a *Token* with a *Marking* to represent that condition of the place holds true. The *Marking* of these *Tokens* can move from one place to a subsequent place base on the horizontal bars that are referred to as *Transitions*. These *Transitions* represent actions that can fire when the places connected towards the transition have conditions that hold true. 30

3.5 A synthesized design of the S-R latch expressed in structural Verilog made of *ContinuousAssignments*. The variable of a *ContinuousAssignment* updates it value when its expression becomes true. This structural Verilog is converted into a decompose form represented in the SBOL data format for describing the genetic circuit. 34

4.1 Four genetic constraints that occur in genetic circuits. **Figure 4.1a** Roadblock: initiation of transcription from the upstream promoter in a tandem promoter is impeded by the presence of a transcription factor bound to the downstream promoter. **Figure 4.1b** Crosstalk: interference of circuit components with each other or the host circuitry. **Figure 4.1c** Signal mismatch: incompatible signal levels of gates composed in series. **Figure 4.1d** Genetic context effects: the same circuit can act differently based on the ordering of neighboring components. 39

4.2 An illustration of the properties that must be set on a *CombinatorialDerivation* for generating transcriptional units to build a preset of genetic logic gates supported in this workflow. The pattern for building multiple transcriptional units with two promoters, two pairs of RBS, CDS, and a terminator is specified as the *Template*. Three *VariableComponents* (*vc*1, *vc*2, and *vc*3) are created for listing possible DNA parts that a component on the transcriptional unit can be assembled from. The DNA parts listed in these three *VariableComponents* are referred to as *Variants*. *VariableComponents* for *vc*1 and *vc*3 have *Operators* (*op*) set to one to indicate that one *Variant* from its list must be selected when constructing a transcriptional unit. Likewise, *VariableComponent vc*2 has *op* set to *zeroOrOne* to indicate that zero or one *Variant* can be selected from its list to generate transcriptional units with tandem promoters and gates with an output reporter included. The *Strategy* is set to *Enumerated* so that all *Variants* listed in each *VariableComponent* must be used to build different variations of a transcriptional unit with one and two input gates. . . . 41

4.3 An enriched design of a transcriptional unit after calling VPR. Four *Interactions* encapsulated within *Modules* are added to the design. These four *Interactions* are an inhibiting interaction for protein X and *Pro*, a production interaction for protein Y and CDS, and two degradation interactions for protein X and Y. The black arrows denote *MapsTo* objects that are used for connecting proteins and DNA parts from the *Modules* encasing the four types of *Interactions* to the components on the transcriptional unit. 43

4.4 Genetic logic gates that are identified and sorted in the gate generation process are *NOT*, *NOR*, *OR*, wired *OR*, *NAND*, *AND*, and *NOTSUPPORTED* gates. A *NOT* gate is identified as a transcriptional unit with a promoter repressed by an input protein X that produces an output protein Y. An *OR* gate is identified as a transcriptional unit with a promoter activated by two input proteins X_0 and X_1 that produces an output protein Y. A wired *OR* gate has identical structure as that of an *OR* gate but the input and output signals, in this case, are all identical. A *NAND* gate is identified as a transcriptional unit with tandem promoters that are both separately inhibited by two input proteins X_0 and X_1 that produces an output protein Y. A *AND* gate is identified as two input proteins X_0 and X_1 forming a protein-to-protein complex to repress Pro. When the protein-to-protein complex is not present, the output protein Y is produced. Because there are several ways to build a genetic logic gate that exhibit the same logic behavior, three different structures for a genetic *NOR* gates are supported. The first structure has a transcriptional unit with two transcription factors repressing a promoter (*Pro*) and producing an output protein Y. The second structure has two forms of input molecules. The first is an input protein X_0 repressing the *Pro* and the second input is a small molecule X_1 forming a complex with the output protein Y. The third structure ressembles the second structure in which the second input signal X_1 forms a complex with the output signal. The difference in the third structure in comparison to the second structure is that X_1 is an input protein rather than a small molecule. Gates that are not identified as a *NOT*, *OR*, wired *OR*, *NAND*, *AND*, and *NOR* gates are classified as *NOTSUPPORTED* Gates. 46

4.5 Extra information included from VPR when identifying *NOR* gates. This type of gate is expected to be identified as the second structure *NOR* gate shown in **Figure 4.4**. Important information might be lost by removing this extra information that is added from VPR. As a result, this type of *NOR* gate is recognized in the gate identify step unchanged as long as this *NOR* gate has the same substructure as the expected *NOR* gate. 47

5.1 A representation of the logic gates decomposed to a graph-based format before performing technology mapping. A 2-input *AND* gate involves attaching two 1-input *NOT* gates as inputs to a 2-input *NOR* gate. A *NAND* gate has a 2-input *NOR* gate with its input and output signals inverted with 1-input *NOT* gates. *NOT* and *NOR* gates, on the other hand, are left in their original form. The graph-based structure shown on the right of each genetic gate represents a *DecomposedGraph*. Circles represent nodes. A root node has no parent. A leaf node has no child. Interactions are used to form connections between nodes. Arrows with pointy heads indicate production and arrows with flat heads are repression. Color represents signal carrier type. A white *DecomposedGraphNode* indicates that there is no signal carrier type assigned. A *DecomposedGraphNode* can have a cost factor that is calculated by DNA sequences. 52

5.2 A figure to illustrate how an SBOL data model is converted into a *DecomposedGraph*. This conversion transforms the given structural specification to a graph-based structure before performing technology mapping so that genetic gates can map to the specification. *NOTSUPPORTED* gates can also use this conversion if a behavioral Verilog was provided and synthesized into a structural representation. 53

5.3 An example of the technology mapping procedure taking in two forms of inputs. The first is a specification describing an AND gate. The second is a library containing five genetic gates. The cost of each gate is calculated by the sum of sequences stored on the *DecomposedGraphNodes*. The color code assigned to each genetic gate's *DecomposedGraphNode* indicates the assignment of a molecule signal. If *DecomposedGraphNodes* have the same color assigned, then this indicates that molecule signals are the same. 58

5.4 The result of the matching step when applied on the *AND* specification. In this step, the library of genetic gates are mapped onto the *AND* specification. Genetic gates that can map to the specification are recorded on the specification's *DecomposedGraphNodes*. Where a genetic gate maps on the specification's *DecomposedGraphNode* corresponds to the gate's root *DecomposedGraphNode*. 59

5.5 An example to show how signal carrier mismatch is determind when covering a genetic gate to the specification. A genetic gate is added to the current solution when the molecule signal on a root *DecomposedGraphNode* of a genetic gate matches the molecule signal assigned to a specification's *DecomposedGraphNode*. In this example, the brown *DecomposedGraphNode* for NOT_2 does not match the yellow *DecomposedGraphNode* assigned on the specification. However, NOT_1 has the same molecule signal that matches the assigned specification's *DecomposedGraphNode* and is thus added to the current solution for covering. 62

5.6 The purpose of EndNodes is used for matching the *DecomposedGraphNodes* for g paired to s. EndNodes are calculated by determining where the root of a genetic gate begins and where the genetic gate's leaf *DecomposedGraphNodes* end on the specification. In this figure, the yellow and blue *DecomposedGraphNodes* indicate where NOR_1 ends on the given specification. The EndNodes calculated in this example is where the yellow and blue *DecomposedGraphNodes* map on the specification *DecomposedGraphNodes*. 63

5.7 An example to show how crosstalk is computed in the covering step. In this example, NOT_2, if added to the current solution, causes crosstalk with an existing genetic gate that has already been selected in the solution. The *DecomposedGraphNodes* causing crosstalk in this example is the tan colored *DecomposedGraphNode* located in the specification conflicting with the tan colored *DecomposedGraphNode* in NOT_2. NOT_1, if added to the current solution, avoids crosstalk because the colors assigned to each *DecomposedGraphNode* in the specification are unique. 67

5.8 A figure showing the result of the technology mapping procedure after calling the covering step on the *AND* specification. Two solutions are derived in this figure after performing the covering step. Signal carrier mismatch and crosstalk are accounted for when generating these solutions. If exhaustive was selected for the running of the covering step, then both solutions from this figure are returned from the technology mapping procedure. If greedy was selected, then the solution returned from this procedure varies based on the number of solutions provided as the input to this covering algorithm. If the number of solutions provided to the method is one, then the solution with a cost of 20 is returned. Greedy selects this solution with a cost of 20 to return from its method because greedy is designed to select gates with the lowest score for covering a specification's *DecomposedGraphNode*. If branch and bound was selected, then the final solution returned from this method is the solution with a cost of 16. This solution was selected from branch and bound because out of both solutions that were found, this solution has the lowest score. 73

5.9 An example of the technology mapping procedure performed on the SR Latch example. The four genetic gates provided as the library are generated from the CELLO SYNBIOHUB collection. The matching step identified NOT_1 and NOT_2 mapped to *Out* and NOR_1 and NOR_2 mapped to $\bar{Q}$ and Q. One solution is produced from the covering step because all three covering algorithms identified that NOT_1 results in signal carrier mismatch if it is added to the solution for technology mapping. 74

5.10 A netlist of the SR Latch represented in SBOL. The genetic gates that were selected in the covering solution are instantiated in the SR Latch as *Modules*. These *Modules* are shown in the figure as NOT_1, NOR_1, and NOR_2. IPTG, aTc, TetR_protein, LacI_protein, and YFP_protein are *FunctionalComponents* instantiated on the *topLevelCircuit* as the input and output signals of these genetic gates. The connections that are formed from a gate's input and output signals to the instantiated signals on the SR Latch are done so through *MapsTo* objects. 78

6.1 **Figure 6.1a** and **Figure 6.1b** are the two areas in the workflow where verifications are performed. Synthesis verification takes the behavioral Verilog and the structural Verilog produced from synthesis to perform simulation. The simulation results are verified to check if they are equivalent in simulation behavior. Genetic circuit verification is performed on the technology mapping procedure. This verification workflow takes the SBOL converted from the structural Verilog and the SBOL netlist produced from technology mapping. Both SBOL representations are compared and simulated within iBioSim by converting SBOL to SBML. The simulation results are compared and checked if they are equivalent in simulation behavior. 81

6.2 Simulation results for synthesis verification. **Figure 6.2a** shows a behavioral Verilog simulation of an SR latch executed in ModelSim. **Figure 6.2b** shows a structural Verilog simulation of an SR latch executed in ModelSim. 82

6.3 Simulation results for synthesis verification generated within iBioSim. **Figure 6.3a** shows a behavioral Verilog to SBML simulation of an SR latch. **Figure 6.3b** shows a behavioral Verilog to LPN simulation of an SR latch. 82

6.4 Simulation results for synthesis verification. **Figure 6.4a** shows a structural Verilog to SBOL simulation of an SR latch. **Figure 6.4b** shows a simulation of an SR latch after running technology mapping. 83

7.1 Mapping of Cello DNA parts used for building transcriptional units. The left table in the figure shows the different combinations that are used for a transcription unit with two promoters. The rows indicate promoters on position 1 and columns indicate promoters on position 2. Green indicates that the pairing is valid and red indicates the pairing is invalid. The right table in the figure shows the list of cassettes. 86

7.2 Behavioral Verilog of a generalized C-element. 87

7.3 The testbench of a generalized C-element. 88

7.4 A genetic sensor that uses filtering and communication to improve its reliability. The logic diagram produced by logic synthesis. It is composed of three gC gates that go high when both inputs are high and go low when the input not marked with a "+" goes low. The output of the second and third gate are connected to the "+" input of the next gate. The detection begins when both IPTG and aTc go high, activating Cell 1. This creates the quorum signal LasI to diffuse to Cell 2, which then activates Cell 2 to produce the quorum signal RhlI. The RhlI signal diffuses to Cell 3 to activate YFP production. However, if IPTG goes low at any point during this chain reaction, the whole circuit resets. 89

7.5 Structural Verilog of a generalized C-element after running ATACS. 90

7.6 A decomposed gC described using *NOT* and *NAND* gates after running Yosys. **Figure 7.6a** represents the decomposed circuit using logic gates. **Figure 7.6b** represents the decomposed circuit using *DecomposedGraph*. 91

7.7 Result of circuit after running technology mapping. **Figure 7.7a** represents a netlist of the genetic circuit. **Figure 7.7b** is a flattened representation of the netlist designed on a plasmid. 92

7.8 The genetic design produced by iBioSim, which is composed of three genetic sequences that can be put onto separate plasmids and transformed into cells to create three cell types. 92

7.9 iBioSim ODE Runge-Kutta simulations (measured in number of molecules) of the genetic sensor demonstrating the filtering behavior. For the circuit outputs to turn on (LasI, RhlI, and YFP for the three cells, respectively), both aTc and IPTG inputs need to be present; however, if the IPTG signal is only briefly present, then the circuit filters this and does not show the YFP output. This behavior can be seen in the four different panels where the IPTG is present for differing amounts of time. **Figure 7.9a** IPTG is present for 10 time units and almost no LasI is produced. **Figure 7.9b** IPTG is present for 100 time units and LasI is produced by the first cell but only a little RhlI is produced by the second cell. **Figure 7.9c** IPTG is present for 300 time units seconds and both LasI and RhlI are produced with a clear delay between the two peaks. **Figure 7.9d** IPTG is present for 750 time units, providing sufficient time for the final circuit output (YFP) to be produced. 94

LIST OF TABLES

5.1 This table summarizes the different existing technology mapping tools. It shows the tool's input specification language, standards supported, the types of library parts used, and the genetic constraints addressed. 79

LIST OF ALGORITHMS

4.1 Gate Identifier 44
4.2 Gate Interaction 45
5.1 Match 55
5.2 isMatch 56
5.3 Covering Algorithm 61
5.4 Cover for 1-input Gate 64
5.5 Cover for 2-input Gate 65
5.6 Update Covering Solution 66
5.7 Crosstalk Implementation 66
5.8 Branch and Bound 69
5.9 Branch and Bound Cover for 1-input Gate 70
5.10 Branch and Bound Cover for 2-input Gate 71
5.11 Generating SBOL NetList 75
5.12 SBOL Netlist Molecule Connections 76

ACKNOWLEDGEMENTS

There are many people I would like to acknowledge for the completion of this dissertation. I want to express my gratitude to my research supervisor, Professor Chris Myers, for allowing me to do research and for providing me invaluable guidance throughout this time. The knowledge I accumulated during my research is vast, and it is attributed to his patience and his willingness to teach, disregarding the times it took for concepts to engrain themselves. He has taught me to never stop asking questions and thereby, has helped me improve my skills to analyze, evaluate, and explain information. He has also introduced me to synthetic biology and systems biology. These fields have been instrumental in my research and have allowed me to make significant contributions. In addition to Professor Chris Myers, I want to thank my supervisory committee, Tara Deans, Priyank Kalla, Nicholas Roehner, and Ken Stevens. They steered me in directions that helped me produce quality work and provided invaluable feedback where this research can expand on.

I am grateful to have worked alongside great colleagues: Curtis Madsen, Nicholas Roehner, Andrew Fisher, Zhen Zhang, Leandro Watanabe, Meher Saminemi, Zach Zundel, Michael Zhang, Pedro Fontanarrosa, Jeanet Mante, Samuel Bridge, and James Scholz. I want to thank these members for the times that we have spent working on projects, playing board games to decompress, and engaging in discussions in and out of the research. I am fortunate to be a part of a supportive team, and I cherish the fond memories that we have built together within this lab.

I would like to thank those who are involved in the development of the software tools that are used in this research. These tools include SynBioHub, SBOLDesigner, iBioSim, and the community representing SBOL and SBML. The work presented in this research could not have been made possible without these developers. I thank them for their hard work and the time that these developers have invested in producing quality products.

Throughout my research, I participated actively in the SBOL community. In this community, I met great researchers and one person I would like to highlight is Dr. Jacob Beal. Since I've joined the community, Dr. Beal has encouraged me to have a voice. Dr. Beal has taught me leadership skills and valued my inputs to the development of SBOL. I have learned a lot from working with Dr. Beal and I would like to thank him for taking time to help me grow as a researcher.

I also have a strong support system that has been a significant influence towards my academic career. I want to thank Leandro Watanabe and his family for their unwavering support. His family has provided a space where I can always go to study and work. He, on the other hand, is an excellent example of showing me what it is like to set goals and achieve them. I value his constructive criticisms because they helped me continue to grow. I know that whatever milestones I achieve, he is always there in the background to encourage me.

I want to thank my siblings, Thi Nguyễn, Yếnnhi Nguyễn, and Khánhly Nguyễn. I thank them for treating my success and my accomplishments as their own. I feel like I can do anything because of their support. I thank them for cheering me on in everything I set out to do.

I would like to thank my parents, Lân Nguyễn and Ánh Trần. I chose to pursue higher education because they taught me its value. I enjoy learning and developed a fondness for the STEM field at a young age because of my parents. They did this by tutoring me in mathematics so that I had a subject where I could do well in school when first immigrating to America. My parents have made many sacrifices during these hard times so that everything I dreamed became a reality. I thank them for all of what I have accomplished and the person I've become.

Last but not least, I would like to thank my uncle and aunt, Khoa Nguyễn and Bích Nguyễn, for providing the opportunity to achieve my American dream. I continue to accomplish my dreams because I can do so. I thank them for providing me this opportunity.

This research is based upon work supported by the National Science Foundation under Grants CCF-1748200 and DBI-1356041. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

CHAPTER 1

INTRODUCTION

Digital circuits have come a long way since the discovery of transistors, which is the basic building block of all digital circuits that exist today. One of the first digital devices was the transistor radio, TR-1, which was composed of only four transistors [1]. Over the years, novel and more complex digital circuits were developed, which required a growth in the number of transistors needed to build these digital circuits. In the early 1970s, Gordon Moore raised an observation to predict the growth rate of transistors known as Moore's Law. According to Moore's Law, the number of transistors in a chip will continue to double every two years. True to Moore's prediction, the number of transistors found in electronic devices today are in the billions, such as the A12 Bionic chip found on the iPhone XS.

The main driving force that enabled advancements in digital circuits was *Electronic Design Automation* (EDA). EDA is composed of software tools that work together to form a workflow for designing and analyzing electronic systems. Before EDA tools were created, engineers had to design their digital circuits by hand. EDA tools were introduced to help eliminate manual labor, such that software tools automate the process of designing electronic systems. The result of EDA software tools has helped increase the complexity of circuit design, and they have also helped validate the functionality of circuits before manufacturing them.

An EDA workflow starts off with a high-level description language that describes the behavior of the digital circuit being designed, as shown in **Figure 1.1**. This high-level description is represented in a hardware description language (HDL), such as Verilog [2] or VHDL [3]. Once the behavioral description of the circuit is specified, the design then goes through logic synthesis, which is a process that translates the behavioral design to a corresponding gate-level design. The gate-level designs then go through a procedure that translates structural design to physical design. An important part of this procedure is a

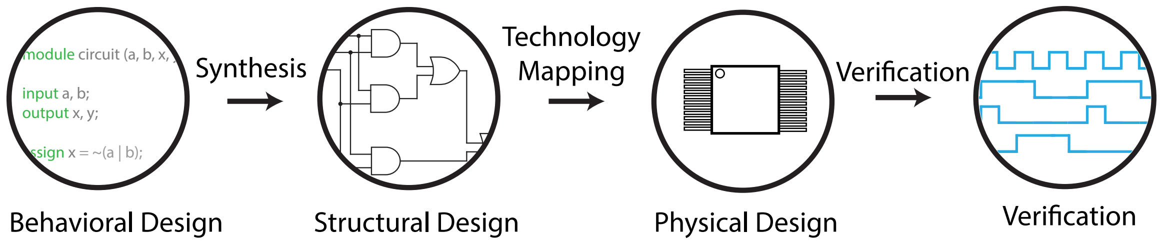


Figure 1.1: A high-level overview of an EDA workflow to build an electronic circuit. First, the behavior of a circuit is described in a high-level descriptive language. The behavioral design is then synthesized to a structural design that describes a gate-level description. Technology mapping is performed to produce a layout of what electronic components are needed to physically build the circuit. Verification is performed to ensure the layout of the physical design meets the requirement of the specification described in the behavioral design.

layer that maps transistor-level gates to implement the gate-level design.

1.1 Genetic Design Automation (GDA)

Synthetic biology is a field that explores the design of *genetic circuits*. Genetic circuits are built from biological components that have been engineered and assembled to perform a specified function within a biological system. The first signs of synthetic biology involved observing how regulatory circuits within a cell respond to environmental cues [4]. When molecular components and their interactions were better understood, it led to the discovery that cellular networks are hierarchically grouped and molecular components can be modularly assembled, such that they can be rearranged or tuned to engineer regulatory networks. This discovery laid the foundation on how genetic circuits are built and brought about the first sequential circuits [5, 6] containing internal states that allow them to map a sequence of input signals to a sequence of output signals. Not long after, a library of combinational circuits [7] producing an output signal solely based on its input signals were also implemented as well.

The construction of these types of circuits stemmed from the idea of building a genetic circuit that replicates the same functionality of an electrical circuit [4]. Similar to electrical circuits, genetic circuits constructed within cells operate by sensing input molecule(s) to produce an output molecule. The signals that are carried across genetic circuits are dependent upon the presence and concentration of molecules that exist within the cell. The

DNA parts that are used to construct a genetic logic gate are represented as hierarchical groupings of DNA sequences.

Since the breakthrough of engineering genetic regulatory networks (GRN), many sequential [5, 8–13] and combinational circuits [14–16] have continued to be developed to build useful applications. In particular, synthetic biologists are designing genetic circuits to produce useful biochemical and pharmaceutical products [17–30], to help cure genetic diseases and create therapeutic bacterial agents [31–43], and to produce plants that can sense and adapt to a wider range of environments [44–50].

For the field to scale up and develop more complex circuits for other applications, synthetic biologists started using principles from EDA to develop *Genetic Design Automation* (GDA) for the systematic design of genetic circuits. **Figure 1.2** shows a high-level GDA workflow. This GDA workflow is similar to the EDA workflow except the physical design generates DNA sequences.

Many computer-aided design (CAD) tools were developed to aid the synthetic biologist in the design of genetic circuits. Today, there are GDA tools that can be found for specification [51–58], design [59–72], build/assembly [73–80], test/analysis [81, 82], data [83–87], simulation [88–99], and sequence editing [100–102].

CAD tools geared towards the design of combinational circuits have been developed [15, 64, 66, 103–105]. `Cello` [15] is an example of such a tool. This tool can be used to design complex circuits consisting of NOT and NOR gates. Many circuits designed in `Cello` have been validated in a wet lab and they have shown promising results.

However, there is a need for a GDA tool designed for sequential circuits and certain barriers exist to develops these tools. First, there is a need for a large collection of well-

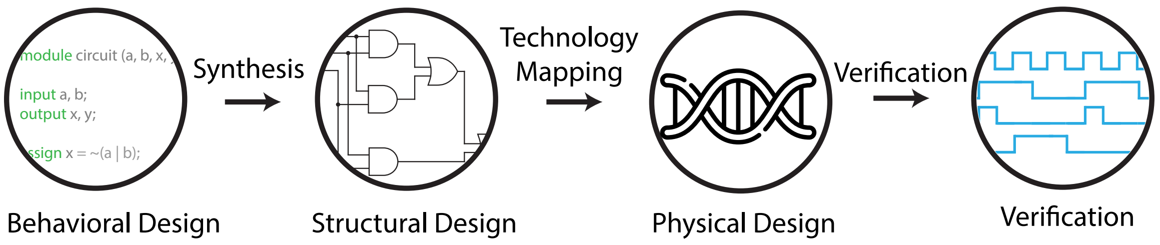


Figure 1.2: This figure shows a high-level GDA workflow. This GDA workflow is similar to the EDA workflow except the physical design generates DNA sequences.

characterized parts in order to construct an abundance of memory circuits [10]. However, creating a memory collection is challenging because memory circuits constructed on transcriptional regulatory networks must exhibit positive or double negative transcriptional feedback loops. This design style limits the number of DNA parts that can be used to construct a memory circuit and as a result, limits the ability for complex circuit to be constructed using memory circuits. Second, design principles and analogies between electronic and biological circuits are required to create effective tools for sequential genetic circuits [10]. Namely, it is important to understand the synchronous and asynchronous styles for designing sequential circuits.

Synchronous designs require the use of a global clock signal to produce a fixed-time schedule for updating the state to store information. Asynchronous designs, on the other hand, determine when to update their state using handshaking protocols. The advantage of designing synchronous circuits is the state of a signal only changes when the global clock is signaled to change the state. This allows signals to stabilize before a new clock cycle passes through and the next operation is performed. However, because synchronous circuits rely on a global clock, the delay is proportional to the time it takes a signal to propagate through the circuit's longest path. In asynchronous circuits, input signals arrive independently and operations happen as soon as all input signals have arrived.

Given the difficulties to create a precise timing reference using biological components and the fact that most biological systems are responsive immediately to changes in environmental conditions, it seems likely that synthetic genetic circuits should follow the asynchronous design paradigm. Nature also makes uses of asynchronous timing. One such example of an asynchronous sequential circuit that occurs in nature is the filtering system utilized by the venus flytrap (*Dionaea muscipula*). It shows different behaviors depending on how often prey touches trigger hairs within a period of time (the circuit is asynchronous as the time element is simply due to molecular decay). At least two trigger events are required to close the trap, and at least three to start producing digestion enzymes [106]. It has been suggested that the memory of the system works by increasing cytosolic calcium levels in a quantized manner correlating with the number of triggers that have been seen [107]. Thus, here the calcium levels are proposed to act as state memory in the asynchronous sequential network.

One GDA tool that was reported to have a framework that can be used to design complex genetic circuits using combinational and sequential circuits and has taken on the synchronous approach was `GeNeDA` [105]. `GeNeDA` was built to resemble an ideal EDA workflow by reusing existing software tools [108–110] from microelectronics to assemble DNA sequences to design genetic circuits. This tool uses Verilog to describe a high-level description of their design. Abstract DNA parts are used from BioBricks to construct the physical components needed to build the specification. Lastly, verification is performed by using the *Systems Biology Markup Language* (SBML) [111] to analyze the behavior of their design. The limiting factor in their design tool is the library that is used to map genetic parts to their specification does not support the traditional memory circuits such as the genetic toggle switch and the repressilator circuits. Instead, the tool used memory modules that were inspired by Hoteit et al. [8]. In the work presented by Hoteit et al., the author noted that existing genetic toggle switches and repressilator circuits behaved more asynchronously rather than synchronously. Although a robust repressilator circuit [9] has been constructed with inputs for tuning oscillation signals, the repressilator circuit still does not have bistable states. Hoteit et al. also presented toggle switches [112–114] that were designed to behave asynchronously because there were no input clock signals that can be connected to a toggle switch to make the circuit perform synchronously. As a result, Hoteit et al. constructed a synchronous memory module that has an input clock using far-red light (FR) to operate on the rising edge of the clock. The result of the Hoteit et al. memory module gave insights on the complexity of the use of a clock in a genetic circuit. Specifically, the FR light used as an input clock signal must be large enough for the circuit to sense when its state should change. If the FR light is removed sooner than the circuit expected, the production of protein in the circuit could be affected. Also, the clock has to turn off for a certain duration before turning back on for an input signal to correctly affect the current state of the circuit.

Needless to say, there is still a need for sequential circuits. Burill et al. constructed a genetic circuit to track cellular memory through cell division when different stimulis are applied to a cell [115, 116]. After cell division, two subpopulation of cells can be formed to track cells that have a high response to the stimulis. Cells that showed high response were shown to exhibit memory based on the effects of growth rate and gene expression of the cells to the stimuli. Their work showed that building memory circuits will provide a better understanding on a cell's heterogeneity and how cells' fate are determined. As a result, future applications are envisioned from building memory circuits to detect and track a subpopulation of cells in a tumor environment and how the subpopulation will contribute to developments of tumor. Burill et al. also noted that in order to create more complex human applications, such as these memory circuits, there is a need for more robust design methodologies to help reduce the number of devices that are built [115, 116].

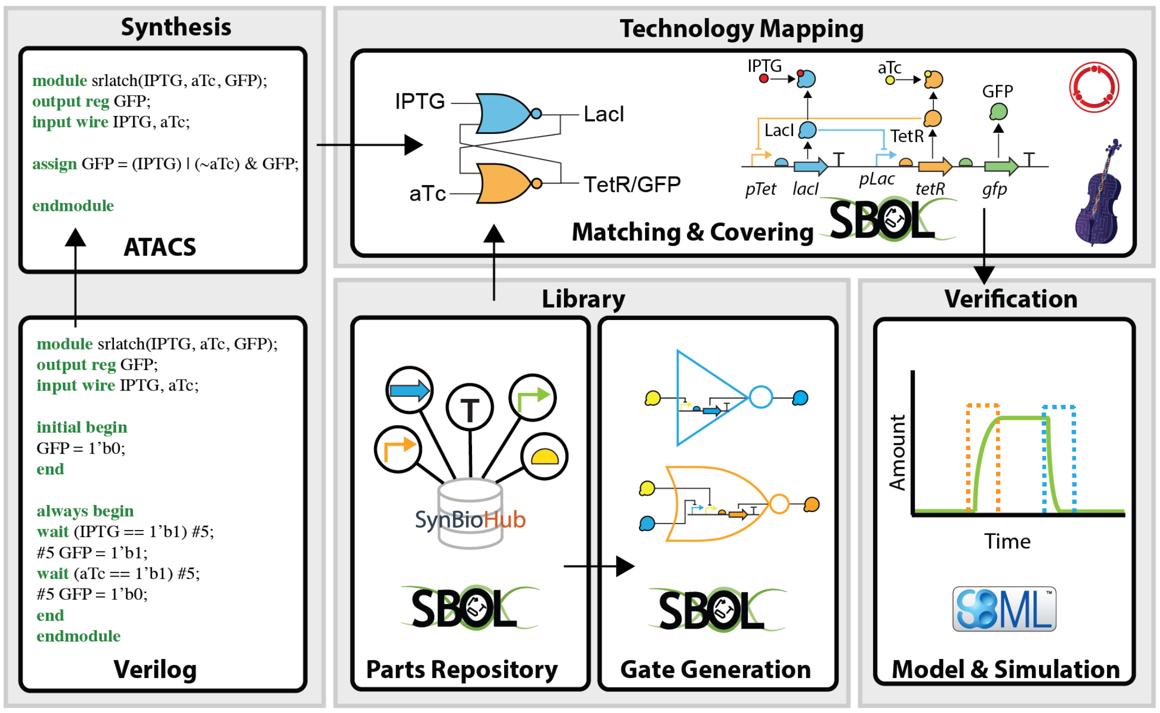
The workflow presented in this dissertation is inspired to take on an asynchronous approach for developing a GDA tool for designing sequential genetic circuits. Including combinational circuits, many of the toggle switches that have been designed to follow the asynchronous nature are well-suited for this realm of work. However, hazards will need to be addressed and solutions on how they are handled are discussed in further details in the later chapters.

1.2 Contributions

As shown in **Figure 1.3**, the contributions of this dissertation are divided into four areas and are highlighted as follows:

- a compiler that translates an asynchronous design described in the Verilog language, and converts it to biological data standards.
- an automated library for gate generation.
- a technology mapping procedure using standardized biological data formats to facilitate the process of designing genetic circuits.
- a verification procedure to validate the workflow.
- a case study for building a sequential circuit.

The main contributions for this dissertation involve the necessary steps for the development of an end-to-end asynchronous circuit design workflow. The workflow presented in this dissertation leverages electronic design styles that have been explored for designing electronic systems. By replicating electronic design styles to model biological systems,



```
module srlatch(IPTG, aTc, GFP);
output reg GFP;
input wire IPTG, aTc;

assign GFP = (IPTG) | (~aTc) & GFP;

endmodule
```

```
module srlatch(IPTG, aTc, GFP);
output reg GFP;
input wire IPTG, aTc;

initial begin
GFP = 1'b0;
end

always begin
wait (IPTG == 1'b1) #5;
#5 GFP = 1'b1;
wait (aTc == 1'b1) #5;
#5 GFP = 1'b0;
end
endmodule
```

Figure 1.3: This workflow is illustrated using a genetic toggle switch design. The workflow starts off with an asynchronous design specified using behavioral Verilog. Next, the ATACS asynchronous design tool synthesizes logic equations represented using structural Verilog. The synthesized circuit is then realized as a physical design by a technology mapping procedure that selects gates from a SBOL encoded gate library stored in a `SynBioHub` repository. In order to verify the workflow, a model is generated and simulations are performed to verify that the design displays the expected behavior.

there are potential outcomes of what this could show when designing genetic circuits and how it could possibly scale up in terms of building complex circuits.

Because of the inherent nature of how biological systems behave, an asynchronous circuit design workflow is proposed. A critical part in design automation is to have a way to describe the behavior of the circuit. Hence, the first contribution of this dissertation is to specify how to model asynchronous circuits in Verilog and how to synthesize the Verilog designs into gate-level designs. In order to realize a physical design from a gate-level design, technology mapping is necessary. A key part of technology mapping is to have a gate library. The second contribution of this dissertation involves an automated procedure to generate a gate library for genetic circuits. The third contribution is a technology mapping procedure for asynchronous circuits. Lastly, this dissertation highlights the proposed workflow with case studies.

A key features that is important for navigating between steps in the workflow is the use of data standards. Data standards have come a long way to increase the result for reproducibility and exchangeability of information between software tools. The biological data standards used in this automated workflow are well-suited for its purpose for designing and performing verification on genetic circuits.

1.3 Dissertation Outline

This dissertation is composed of seven themed chapters.

Chapter 2 provides the necessary background on the different areas that are needed to build a technology mapping workflow for asynchronous genetic circuit designs. First, concepts for constructing a genetic circuit are discussed to get an understanding of how high-level descriptions can map into physical biological components. Then, biological data standards are discussed to show how information about genetic circuits is formally represented throughout the different stages of the workflow. Finally, the chapter discusses software tools that are used in the workflow.

Chapter 3 focuses on understanding how the specification for a sequential circuit is built from a high-level description and how the specification is synthesized to a gate-level circuit. A general template to describe the behavior for the desired sequential circuit is discussed in order to correctly build a specification that follows the asynchronous design style. This chapter also discusses what performing synthesis entails and how a gate-level circuit translates into a genetic regulatory network to perform technology mapping presented in Chapter 5.

Chapter 4 describes how transcriptional units are assembled to form a library of logic gates that are needed to perform technology mapping. An automated gate generation procedure is presented to build genetic gates from transcriptional units. A gate identification procedure is also described in this chapter to classify a gate's logic behavior and the different structure that a genetic gate can take on.

Chapter 5 shows the full process to map a library of gates constructed in Chapter 4 to build the corresponding circuit of a synthesized specification that is obtained from Chapter 3. This chapter describes a Boolean decomposition procedure performed on the specification and the library gates before technology mapping occurs. Then, the technology mapping procedure is broken into two separate sections. The first is a matching step that is used to filter from the library of genetic gates that can map to the specification. The second is a covering step that generates possible solutions from a list of genetic gates for constructing the specification.

Chapter 6 presents a verification procedure performed at different stages in the design process. This chapter describes how each step in the workflow is verified to ensure that the behavior of the design remains consistent in each of stage. This verification procedure is split into two areas. The first verification procedure shows the process for verifying behavioral to structural Verilog. The second verification procedure shows the process for verifying a structural representation of a synthesized genetic circuit design to a netlist of genetic gates.

Chapter 7 goes over a case study to demonstrate how the proposed workflow operates. First, this chapter presents a library of genetic gates created for testing the circuit. Then, this chapter goes through the step by step procedure for designing the intended genetic circuit.

Finally, Chapter 8 summarizes the accomplishments that were presented and discusses future directions for this work.

CHAPTER 2

BACKGROUND

This chapter presents the background information needed to understand the proposed asynchronous genetic circuit design workflow. This chapter is composed of four sections, where each section provides the necessary background in a specific topic. That is, Section 2.1 discusses the differences between synchronous and asynchronous styles for designing genetic circuits. Section 2.2 defines terminologies of a genetic circuit and demonstrates how they behave similarly to that of a logic circuit. Section 2.3 goes over the biological data standards needed for representing genetic circuit designs and modeling genetic circuits that can be used for simulating the designs needed for verification. Lastly, Section 2.4 goes over the features of an existing GDA tool relevant to this dissertation and how it is leveraged to build an end to end workflow for designing asynchronous genetic circuits.

2.1 Asynchronous Circuit Design

Electronic circuits based on digital systems rely on Boolean logic as an abstraction layer. Such abstraction allows signals to be either in a HIGH or LOW state, which helps reason about the structure of the circuit design. There are two types of digital circuits. One is combinational circuits, where the input signals map directly to the output signal. The other is sequential circuits. Unlike combinational circuits, sequential circuits have an internal state that allows them to map a sequence of input signals to a sequence of output signals.

Sequential circuits can be designed synchronously, asynchronously, or a hybrid between the two. Synchronous circuits operate on a global clock used to produce a fixed-time schedule for updating states. However, because synchronous circuits rely on a global clock, the clock rate is determined based on the critical path (i.e., longest path from inputs to an output). As a result, there is no best-case or average-case timing performance [117] when designing synchronous circuits.

Some key challenges in synchronous circuit design include dealing with clock distribution [118–121] and the corresponding power consumption [117] for the clock network. Such challenges do not apply for asynchronous circuits. Asynchronous circuits perform operations without the use of a clock signal. In asynchronous circuits, the states are updated by using *handshaking protocols* as a mechanism to synchronize communication and *data encodings* to represent information. Handshaking protocols have *request signals* to indicate there is information to be sent and *acknowledge signals* to indicate that the information has been received. One benefit of asynchronous circuits is that they can exploit average-case performance, since the states are updated as soon as possible after the input signals arrive.

In digital circuits, there are cases when a signal switches erroneously due to a *glitch* (a spontaneous transition of a signal going HIGH to LOW or LOW to HIGH). Glitches on synchronous circuits are not a major concern as long as the signal is stable before or after the clock signal is sampled to produce a valid result [117]. However, this puts more reliability on carefully controlling the global clock to ensure glitches are filtered. However, glitches in asynchronous circuits are problematic. In asynchronous circuits, glitches, also referred to as *hazards*, happen when a signal that is supposed to remain stable momentarily changes value, or a signal that is supposed to change does so non-monotonically. Unless the circuit is validated to be hazard-free, unexpected input signals could result in invalid circuit outputs [117].

The S-R latch and the clocked S-R latch shown in **Figure 2.1** illustrate the difference between synchronous and asynchronous circuits. A S-R latch is a circuit designed asynchronously for capturing information when it senses a change in its input signals. A clocked S-R latch is a level-sensitive circuit designed synchronously for capturing information based on a clock signal (i.e., only change state when clock is high). Set, S, and Reset, R, are input signals used to control what data are captured in the circuit. The output signals, Q_{A}, $\bar{Q}_{\mathrm{A}}$, Q_{S}, and $\bar{Q}_{\mathrm{S}}$ are used to show what data value is currently stored within the circuits. As shown in the waveform, both circuits behave in the following manner. Q_{A} produces a HIGH signal when S is set to HIGH. S and R are never allowed to be HIGH at the same time but can both be LOW at the same time. When both input signals are LOW, then the previous output state is returned. When R is set to HIGH, $\bar{Q}_{\mathrm{A}}$ goes HIGH. The main difference between the S-R latch and the clocked S-R latch is the use

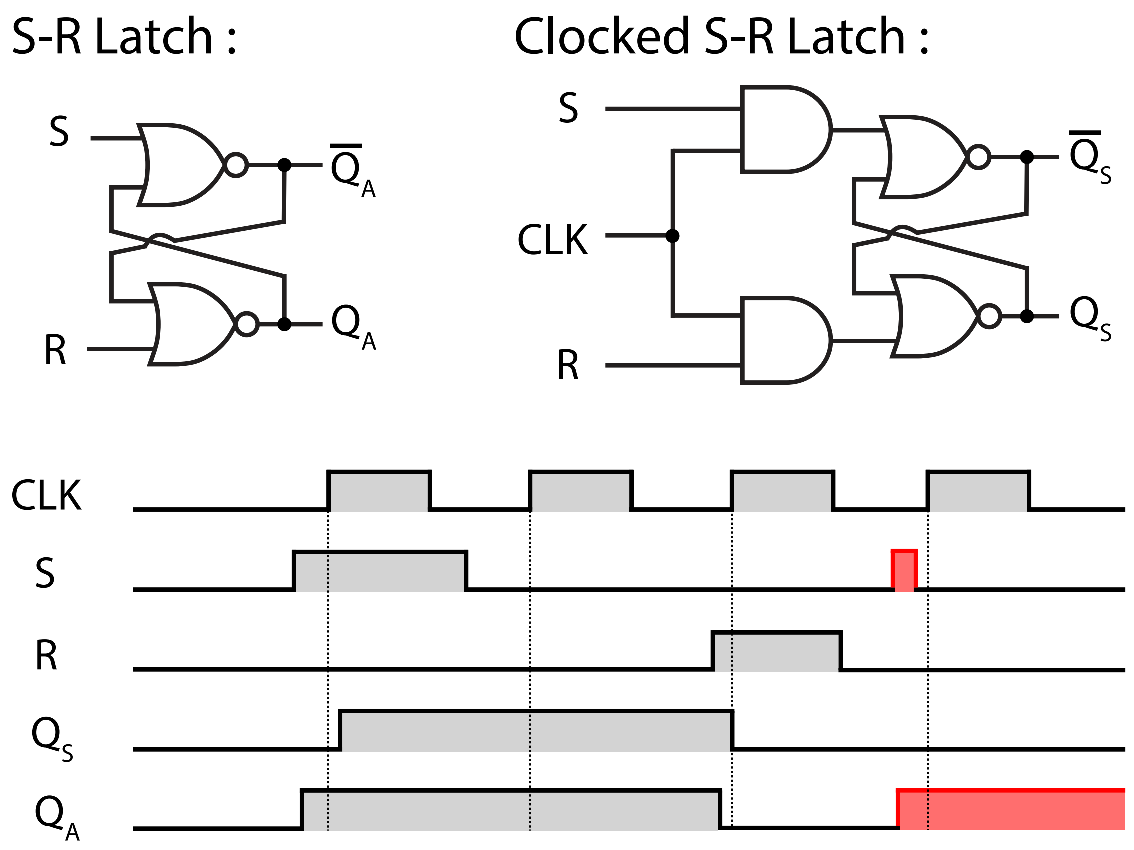


Figure 2.1: This figure shows the difference between an asynchronous and a synchronous S-R latch and a waveform illustrating the behavior of both circuits after they have been initialized for simulation. The S-R latch is a level-sensitive circuit designed asynchronously for storing information when there is a change on the input signals. The clocked S-R latch is a level-sensitive circuit designed synchronously for storing information based on a clock level. Set, S, and Reset, R, are input signals used for setting the circuit's data value. $\bar{Q}_A$ is the inverse signal of Q_A and are both represented as output signals to show what data value is currently stored within the S-R latch. The same can be said for Q_S and $\bar{Q}_S$. $\bar{Q}_A$ and $\bar{Q}_S$ are simplified in the waveform as the LOW state of Q_A and Q_S. The behavior of the clocked S-R latch is identical to that of the S-R latch except that the output signal is governed by a clock, CLK, signal. In other words, the output signals for Q_S and $\bar{Q}_S$ are designed in this circuit to update when CLK is HIGH. The red signals that go HIGH on S is a glitch that appeared for a short duration and causes Q_A to go HIGH unexpectedly. The clocked S-R latch is able to avoid this glitch because the input signal was not stable before CLK could capture this information. The S-R latch, however, encounters the glitch as a change in the input signal and produced an invalid output signal.

of a CLK signal in the clocked S-R latch. Specifically, the output signal on the clocked S-R latch is only updated when the CLK signal is HIGH. It is important to note that, in synchronous circuits, edge-sensitive flip-flops are more commonly used than clocked S-R latches. However, clocked S-R latches are used in this context because they are easier to explain.

Glitches can occur on sequential circuits. The circuit could possibly latch onto the a glitching signal or it could also cause the circuit to go metastable. It depends on how long the glitch persists and the relative delay of the circuit. As an example, the red signal that went HIGH on S for a short duration in **Figure 2.1** illustrates a glitch. For synchronous circuits, this glitch can be avoided because the use of a clock signal ensures that the input signals must be stable before capturing information. On the other hand, an asynchronous circuit does not see this glitch signal as an unexpected signal that should be ignored. In this specific case, the glitch on the input signal, S, causes the output signal, q, to latch onto the HIGH state when it was not expected to. This example shows that it is critical to eliminate hazards in asynchronous circuit designs. Therefore, it is important to account for hazards, such as the one shown in this example, when designing circuits asynchronously to ensure correct behavior. This can be accomplished in a couple of ways. Redundant logic can be added onto the design, timing analysis can be performed on the design, or delays can be inserted to limit the behavior of a circuit.

Designing hazard-free asynchronous designs is challenging and a number of CAD tools have focused on supporting asynchronous designs to be hazard free [122] . ATACS [122] is an example of such a tool. ATACS is an asynchronous logic synthesis tool that derives Boolean logic equations that implement a specified design. ATACS creates a state graph from the given labeled-Petri net [123] (LPN) model.

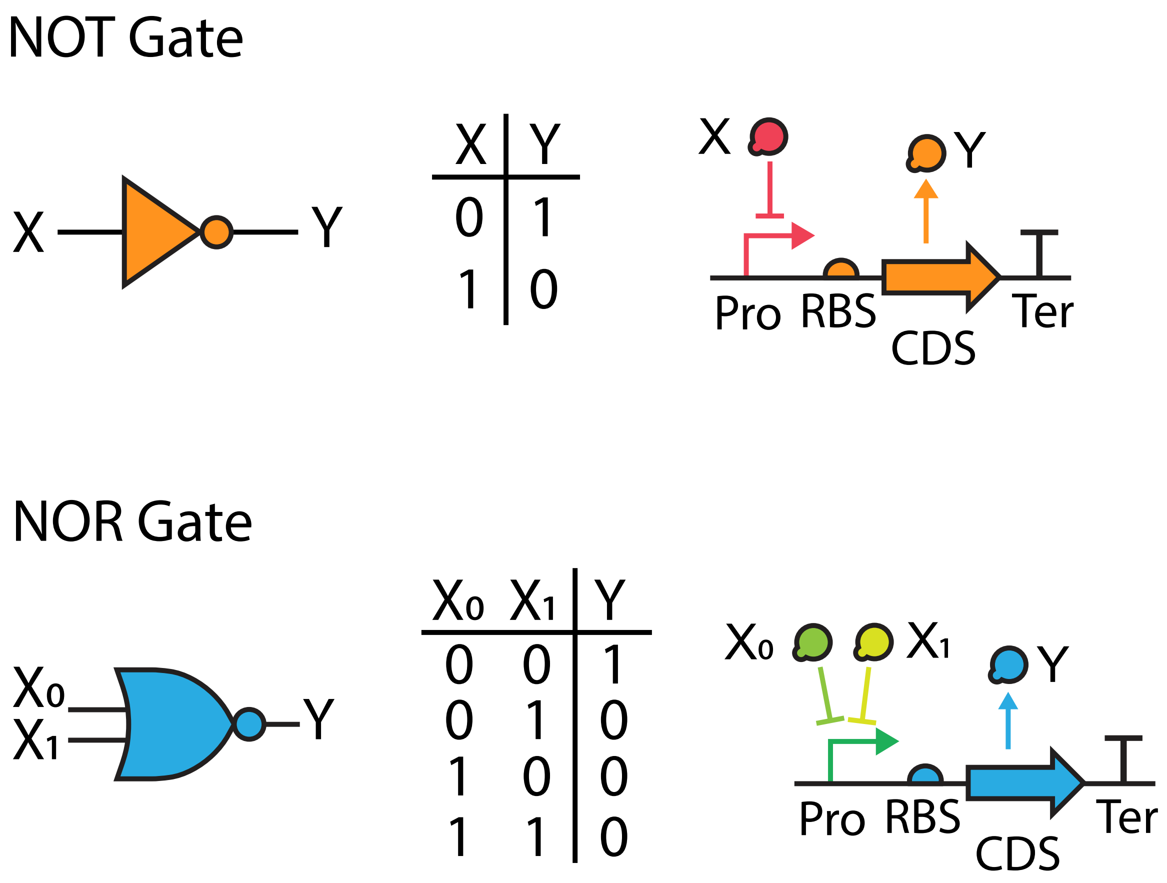
To do so, one area of work proposed in this dissertation is to convert a high-level asynchronous design, described in Verilog, into its equivalent LPN model. Once the high-level description of the design is modeled as an LPN, it is translated into a state graph within ATACS to perform synthesis. The state graph is analyzed to determine the logic necessary to implement any state and output signals. Synthesis from ATACS produces hazard-free Boolean logic functions described in a structural Verilog representation. The structural Verilog representation produced from ATACS can then be used for technology mapping to derive the asynchronous genetic circuit.

2.2 Genetic Circuits

Many biological mechanisms have emerged to engineer genetic circuits [124, 125]. The most popular mechanism is transcriptional regulation. The main advantages of using transcription regulation includes modularity of the circuit so that it can be reused for building more complex circuits and orthogonality of the circuit to function independently from interfering with other functions within a cell. Genetic circuits designed and built using this mechanism have been documented in literature and can be accessed through online databases. For these reasons, the work demonstrated in this dissertation uses the concept of transcriptional regulation for designing genetic circuits.

Genetic circuits using transcriptional regulation are made of *transcriptional units*. Transcriptional units store information on a complex molecular structure called *deoxyribonucleic acid* (DNA). The central dogma of biology is that DNA is used to create RNA (*ribonucleic acid*) through a process called *transcription*. The RNA is in turn used to create proteins though a process called *translation*. Some of the proteins created, called *transcription factors*, can regulate (i.e., *activate* or *repress*) further protein production. The sequence that is directly used to create a protein is called the *coding sequence* (CDS). Not all of a genetic sequence is directly used in the creation of proteins. For example, there are sequence regions called *promoters* where transcription is initiated and that include binding sites for transcription factors allowing them to regulate the speed of transcription. *Ribosome binding sites* (RBS) are sequences where *ribosomes* can bind to the RNA to initiate translation. Finally, there are *terminators*, which are regions of the DNA that indicate where transcription should stop.

These transcriptional units can interact with other transcriptional units through transcription factors. Complex behavior can be achieved by combining transcriptional units. For example, transcriptional units can be used to construct genetic logic gates as shown in **Figure 2.2**. A genetic *NOT* gate is built with one transcriptional unit composed of a promoter, *Pro*, an RBS, a CDS, and a terminator, *Ter*. The transcription factor, X, represents the input protein to the genetic gate and the encoded protein translated from the CDS represents the output protein from the gate. The promoter, *Pro*, is repressed by a transcription factor, X, such that, when it binds to the promoter, no protein is produced. If X is absent, then the output protein Y is produced. This genetic circuit behaves like a *NOT* gate such



X	Y
0	1
1	0

X_0	X_1	Y
0	0	1
0	1	0
1	0	0
1	1	0

Figure 2.2: A *NOT* gate and a *NOR* gate represented as a genetic circuit. The symbol of the logic gate is shown on the left. A listing of all possible combinations of the gate's input signals (X, X_0, and X_1) mapped to the output signal, Y, is shown in the Boolean table at the center. A design of a genetic circuit that behaves similarly to that of its logic gate are shown on the right. These genetic circuits are made of transcriptional units that undergoes the process of transcription and translation to transform DNA to protein. A transcriptional unit is composed of promoters, *Pro*, ribosome binding sites, RBS, coding sequences CDS, and terminators, *Ter*. The encoded protein translated from CDS is illustrated in both genetic circuits as protein Y and are represented as the output protein for the circuit. The production of the output protein Y can be controlled by proteins that can bind to the the transcriptional unit through repression or activation of the promoter. These proteins are referred to as transcription factors acting as input proteins to a genetic circuit and are represented in this example as X, X_0, and X_1. X, repressing *Pro* in the *NOT* gate example, prevents the production of the Y protein. If X is removed, this allows for protein Y to be produced. The same concept applies for building a genetic circuit that behaves similarly to a *NOR* gate. The presence of X_0 and X_1 represses *Pro*. As long as one input protein is present, the output protein Y cannot be produced. If both input proteins are removed, then protein Y is produced.

that the output signal is inverted based on the presence of the input signal.

A *NOR* gate is a universal logic gate that can be used to realize any Boolean function. **Figure 2.2** shows a genetic logic gate that behaves similarly to a *NOR* gate. This type of genetic logic gate is built from a transcriptional unit composed of a promoter, *Pro*, an RBS, a CDS, and a terminator, *Ter*. The production of protein Y, in this case, is controlled by two input proteins X_0 and X_1 repressing the Pro. As long as one of the promoters is repressed, no output protein is produced from the circuit. If both of the promoters are not repressed, then the output protein Y is produced. This genetic circuit behaves similar to that of a *NOR* gate because an output signal is produced only when both input signals are not present. Because this type of *NOR* gate is versatile for building any Boolean functions, this gate, along with the genetic *NOT* gate, is used in this proposed dissertation for designing an asynchronous genetic circuit. The use of these two genetic gates is discussed in more detail in a later chapter.

Another form of genetic circuit that is used for the work proposed in this dissertation are memory devices that are needed for sequential circuits. Memory devices are constructed using feedback loops. A well-known memory device that has been built for useful applications in synthetic biology is the genetic toggle switch [5]. As shown in **Figure 2.3**, a genetic toggle switch produces two primary proteins (LacI and TetR). The *green fluorescent protein* (GFP) serves as an output reporter that indicates when the TetR protein is produced. The LacI and TetR proteins mutually repress each other through negative feedback loops that control the production of the opposing protein. The production of the LacI protein causes the pTet promoter to be repressed and prevents the production of TetR. Similarly, the production of the TetR protein causes pLac to be repressed and prevents the production of LacI. IPTG and aTc are small molecules used as input sensors to control the production of LacI and TetR. When IPTG is applied, it forms a complex with the LacI protein and acts as a repressor to the LacI protein. If the LacI protein is repressed, this allows the TetR protein to produced. If aTc is applied, the small molecule forms a complex with TetR and represses the production of TetR. The repression of TetR allows for the production of LacI. If both IPTG and aTc are removed, then the genetic toggle switch produces whatever output protein that was synthesized in the previous state.

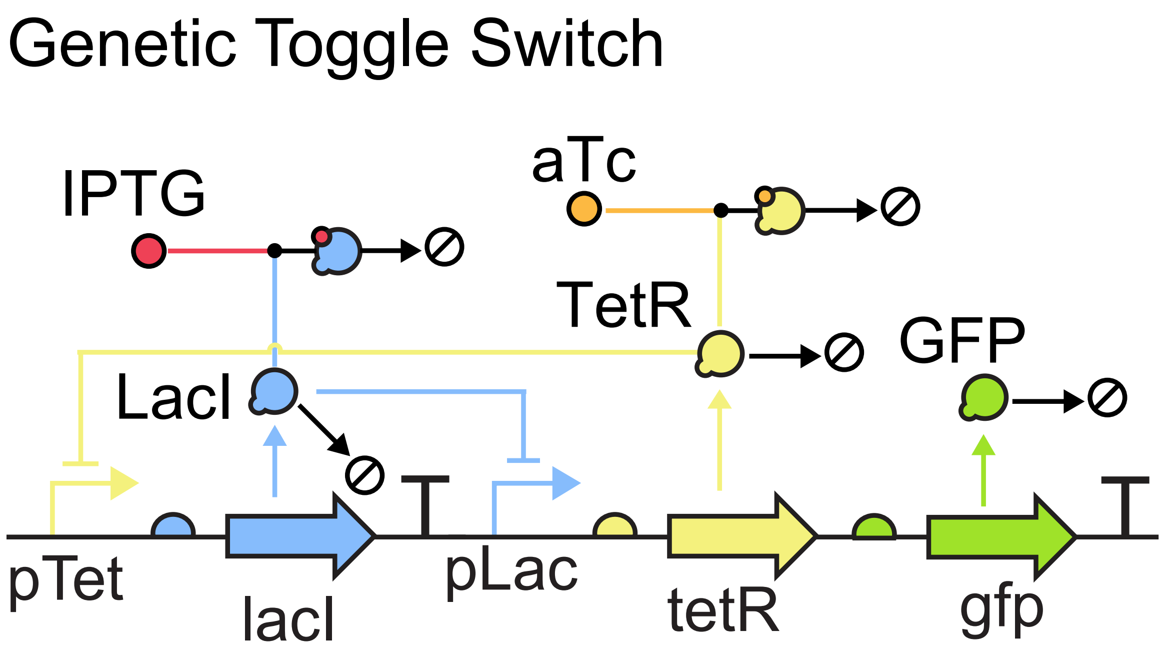


Figure 2.3: A genetic toggle switch is a genetic circuit that behaves similarly to that of an S-R latch. The genetic toggle switch shown in this example is composed of a transcriptional unit that primarily produces a LacI protein and a TetR protein. The *green fluorescent protein* (GFP) serves as an output reporter to indicate when the TetR protein is produced. Both the LacI protein and the TetR protein can mutually repress each other through the negative feedback loops that affect the production of its counter protein. This means that, when a LacI protein is produced, the pLac promoter is repressed, thus preventing the production of a TetR protein. Likewise, if a TetR protein is produced, the pTet promoter is repressed and no LacI protein is produced. The LacI protein and the TetR protein can be controlled by IPTG and aTc to switch between states of the two proteins. IPTG and aTc are small molecules that act as input sensors for controlling the production of the LacI and TetR protein. When IPTG is applied, it forms a complex with the LacI protein and acts as an inhibitor to the LacI protein. If the LacI protein is repressed, the pLac promoter is free to synthesize the TetR protein. Similarly, when aTc is applied, it forms a complex with the TetR protein and acts as an inhibitor to the TetR protein. If the TetR protein is repressed, the pTet promoter is free to synthesize the LacI protein. If both input sensors for IPTG and aTc are removed, then the genetic toggle switch produces whatever output protein that was synthesized in its previous state.

2.3 Biological Data Standards

In order to develop software tools that can automate the genetic circuit design process, data standards must be used to transfer information between the different tools used in each design step. As such, data standards play a big role in the workflow described in this dissertation. In particular, this workflow uses two biological data standards, the *Synthetic Biology Open Language* (SBOL) [126, 127] and the *Systems Biology Markup Language*

(SBML) [111].

2.3.1 Synthetic Biology Open Language

SBOL has emerged as an international standard to describe and exchange structural and qualitative information about genetic circuits. This standard is useful to specify designs in terms of constituent components. Using SBOL, the order and sequences of biological components in a design can be captured, and these designs can be hierarchically reused in more complex designs. Importantly, SBOL supports capturing molecular interactions between these components.

Aside from the SBOL data model, SBOL also supports the use of SBOL visual. SBOL visual represents a collection of graphical symbols that have guidelines and suggestions for capturing information about the functional and structural relationship described in the SBOL data model. SBOL visual can be used for representing genetic circuits and biological parts.

A demonstration of how SBOL is used to represent a genetic toggle switch is depicted with SBOL visual graphical symbols in **Figure 2.4**. This example is constructed hierarchi-

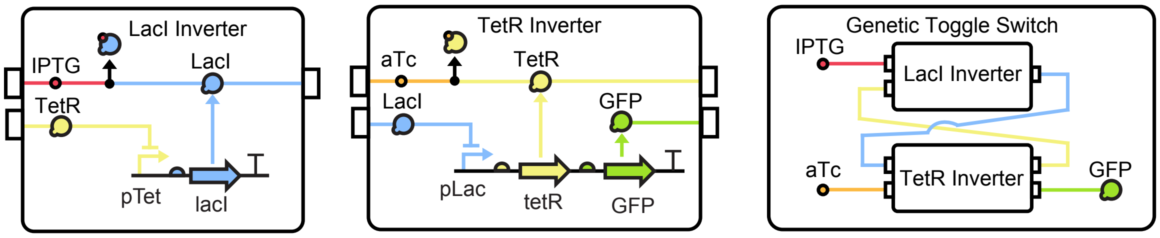


Figure 2.4: A genetic toggle switch modeled hierarchically using SBOL visual symbols to represent the SBOL data model. The circuit and its inverters are described using SBOL *ModuleDefinitions*. Instantiation of the LacI Inverter and TetR Inverter as referred to in SBOL as *Modules*. These *Modules* are connected through the use of SBOL *MapsTo*. The DNA, small molecules, proteins, and their complexes that are shown in this example are described in SBOL as *ComponentDefinitions*. Instantiations of these *ComponentDefinition* within a *ModuleDefinition* are referred to as *FunctionalComponents*. *ComponentDefinitions* can also be instantiated onto other *ComponentDefinitions* as SBOL *Components* to construct a transcriptional unit that is shown in the LacI Inverter and TetR Inverter. SBOL *Interactions* are used to specify the relationship between *FunctionalComponents* such as the forming of a complex between aTc and TetR and IPTG and LacI. SBOL *Interactions* are used in the LacI Inverter and TetR Inverter to represent the inhibition of TetR repressing pTet and LacI repressing pLac. SBOL *Interactions* are also used in both inverters to indicate the production of the LacI protein, TetR protein, and GFP protein.

cally, where the genetic toggle switch is built from a TetR inverter and a LacI inverter using *ModuleDefintions* and *Modules*. Specifically, the genetic toggle switch, TetR inverter, and LacI inverter are repesented as *ModuleDefinitions* and *Modules* are used in the genetic toggle switch to refer to the two inverters. The *Modules* for the LacI inverter and TetR inverter have *MapsTos* used for linking the LacI protein, TetR protein, and GFP protein to their corresponding components in the genetic toggle switch. The components shown in this example with type DNA, RNA, proteins, small molecules, and complexes are represented as *ComponentDefinitions* and are instantiated within *ModuleDefinitions* as *FunctionalComponents*. *ComponentDefinitions* can also be instantiated in other *ComponentDefinitions* as *Components*. In this example, the promoters, RBS, CDS, and terminators found in both inverters are created as individual *ComponentDefinitions*. These DNA parts are joined on a *ComponentDefinition* by referring to the DNA parts as *Components* to represent a complete transcriptional unit. The sequences on each DNA part are referenced on their *ComponentDefinition* as a *Sequence* object.

Interactions are represented in *ModuleDefinitions* to show the relationship between *FunctionalComponents*. An *Interaction* has *Participants* that refer to the *FunctionalComponents* that are involved in the Interaction. In this example, *Interactions* are represented in the genetic toggle switch to show complex formation between IPTG and LacI and aTc and TetR. The *Participants* for forming the IPTG-LacI complex are the IPTG small molecule, the LacI protein, and the complex IPTG-LacI molecule. The same pattern can be expected for representing *Interactions* in both inverters. Specifically, *Interactions* are represented in both inverters to show the tetR CDS producing the TetR protein, the gfp CDS producing the GFP protein, and the lacI CDS producing the LacI protein. *Interactions* are also represented to show the LacI protein inhibiting a pLac promoter in the LacI inverter and the TetR protein inhibiting a pTet promoter in the TetR inverter. The *Participants*, in this case, are the proteins and DNA parts.

There are also other features of the SBOL data model that are useful for representing genetic circuits. The SBOL data model can support different variants of genetic parts assigned to a transcriptional unit through *CombinatorialDerivation*. Information that does not have a direct mapping to the SBOL data object can be annotated through *GenericTopLevel* or *Annotation*. The cycle to design, build, and test a genetic circuit can be recorded using the *Activity*, *Agent*, *Plan*, and *Implementation*.

2.3.2 Systems Biology Markup Language

The *Systems Biology Markup Language* (SBML) [111] is a standard for behavioral models of biological systems at the molecular level and is supported by more than 280 tools, enabling researchers to create, annotate, simulate, and visualize biological models. A SBML *Model* is primarily used for reaction-based models that are composed of a number of chemical *Species* (i.e., proteins, genes, etc.) and *Reactions* that transform these *Species*. *Species* that are consumed by a *Reaction* are called *Reactants*, *Species* that are produced from a *Reaction* are called *Products*, and *Species* that participate in a *Reaction* but neither are consumed or produced (e.g., catalysts) are called *Modifiers*. *Reaction* rates are controlled by *KineticLaws*. *KineticLaws* are typically influenced by rate constants that are represented using *Parameters* (i.e., named variables in a model). While *Reactions* are used to compute the continuous dynamics of *Species*, *Events* can be used to make discrete changes in the state of a model. Events take place when their corresponding *Trigger* condition is evaluated from false to true. Triggered events can be executed immediately or executed in the future through the use of a *Delay* assignment. When an event is executed, its *EventAssignments* are computed to update the state of the model.

SBML Level 3 Version 1 [128] has package extensions [129–132] that generalize the use of the standard even further. The hierarchical composition package (comp), in particular, is a useful extension important for the workflow presented in this dissertation. By using comp, hierarchical models can be constructed through the use of *SubModels*. A top-level model can instantiate multiple *SubModels*. Each *SubModel* refers to a *ModelDefinition*. The *SubModel* can also reference an *ExternalModelDefinition* that is associated with an external file. Connections between *SubModels* can be made through *Replacements* and *ReplacedBys*.

A good illustration for representing SBML through *Systems Biology Graphical Notation* (SBGN) glyphs is shown in **Figure 2.5** for modeling a hierarchical representation of a genetic toggle switch. In this example, the genetic toggle switch is represented as a *Model*. The LacI inverter and TetR inverter are represented as *ModelDefinitions* and are instantiated as *SubModels* in the genetic toggle switch Model. *Replacements* are used to connect the TetR protein, GFP protein, and LacI protein in both inverters to the genetic toggle switch. The

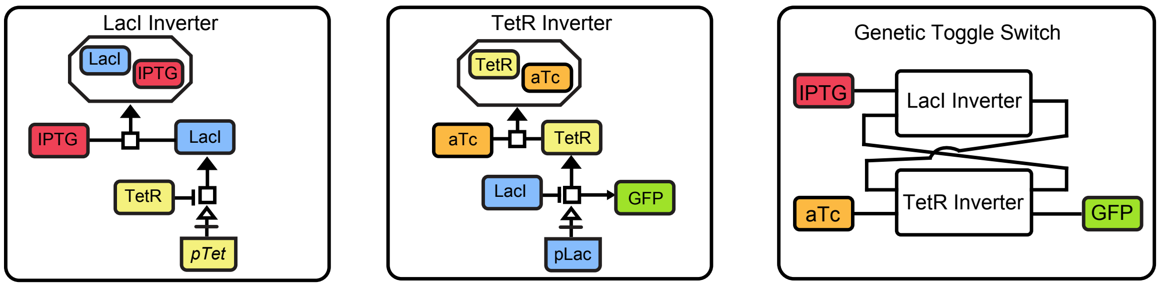


Figure 2.5: A genetic toggle switch modeled hierarchically and is visualized using SBGN graphical symbols to describe the SBML data model. The Genetic Toggle Switch is represented as an SBML *Model*. The LacI Inverter and TetR Inverter are created as SBML *ModelDefinitions* and are instantiated inside of the Genetic Toggle Switch as *SubModels*. *Replacement* and *ReplacedBy* are used for connecting the elements within the Genetic Toggle Switch and the inverters. Small molecules, proteins, complexes, and nucleic acid that are shown in this example are represented as SBML *Species*. The forming of a complex between aTc and TetR and IPTG and LacI is described as SBML *Reactions*. In these type of *Reactions*, the complexes are referred to as the *Products* and the small molecules and the protein used to form the complex are referred to as *Modifiers*. SBML *Reactions* are also used to describe the LacI protein, TetR protein, and GFP protein produced when TetR protein is not inhibiting the pTet promoter and the LacI protein is not repressing the pLac promoter. The TetR protein repressing the pTet promoter and the LacI protein repressing the pLac promoter are referred to as *Modifiers* in their *Reactions*. The LacI protein, TetR protein, and GFP proteins produced in their *Reactions* are represented as *Products*.

DNA, small molecules, proteins, and complexes found in the genetic toggle switch and the inverters are represented as *Species*. Aside from degradation *Reactions* that were omitted in this example to simplify the model, there are two *Reactions* to represent the forming of a complex between aTc and TetR and IPTG and LacI. The complex reactions, aTc, TetR, IPTG, and LacI represent the *Reactants* and aTc-TetR and IPTG-LacI represent the *Products*. There are also two *Reactions* created to model the production of the LacI protein when the TetR protein is not inhibiting the pTet promoter the TetR protein and GFP protein produced when the LacI protein is not repressing the pLac promoter. In these particular cases, the TetR protein repressing pTet and the LacI protein repressing the pLac promoter are referred to as *Modifiers* in their *Reactions* and the LacI protein, TetR protein, and GFP protein produced from their *Reaction* are referred to as *Products*.

2.4 iBioSim

There are GDA tools that exist for putting together the different elements of designing a genetic circuit. One tool that incorporates most of the features that are proposed in this dissertation is iBioSim. iBioSim is a GDA tool for the modeling, analysis, and design of genetic circuits [88, 133]. A high-level illustration of the key features of iBioSim is shown in **Figure 2.6**.

A genetic circuit design in iBioSim begins by using the SBOLDesigner [102] tool to construct a SBOL design by selecting genetic parts from the SynBioHub part repository. SBOLDesigner is an intuitive sequence editor tool that is incorporated into iBioSim as a plugin. This feature facilitates model-based design of genetic circuits by providing the means to construct new designs from existing modeled parts.

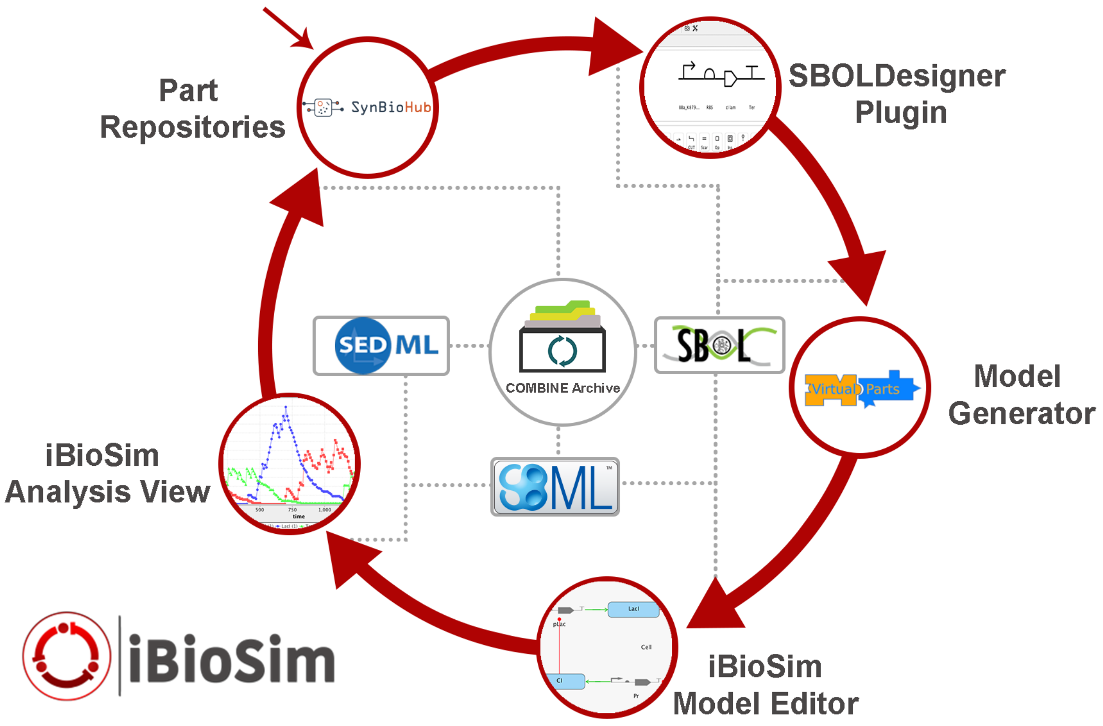


Figure 2.6: This is a high-level diagram of the genetic circuit design workflow supported by iBioSim. The red arrows indicate the flow between the different software components and dotted lines indicate the output of each step that is then used by the proceeding software component in the workflow. First, genetic parts encoded using SBOL are fetched from SynBioHub using the SBOLDesigner plugin to construct the DNA-level design encoded using SBOL. Next, the DNA design is augmented with interaction data using the Virtual Parts model generator, and the functional SBOL is converted into an SBML model. The resulting mathematical model can then be refined and parameters configured using iBioSim's model editor. The SBML model can be analyzed in iBioSim as described by an associated SED-ML document. The data created for the SBOL parts, the SBML model, and the analysis can be shared and documented by uploading these artifacts to SynBioHub as a COMBINE archive.

The VIRTUAL PARTS REPOSITORY (VPR) model generator is utilized to obtain *interaction* data, as described in [134, 135], from the SYNBIOHUB data repository to add functional information to the SBOL description. For example, it adds the proteins that act as *transcription factors* for the *promoters*, as well as their *coding sequences* in the DNA-level design. These *protein components* are coupled with the *DNA components* constructed by SBOLDESIGNER along with their interactions into functional *module definitions*. Next, an SBOL to SBML converter [136] can be applied to translate the structural and functional information of the corresponding SBOL into a quantitative model expressed in SBML. Since SBOL is used to represent qualitative models, the quantitative information required by SBML is inferred [136]. However, this SBML model can then be further refined and model parameters added using IBIOSIM's model editor. Any changes made can be mapped back to SBOL using the SBML to SBOL converter [137]. Simulation is used to verify if the design is behaving as expected. Since one of the goals of IBIOSIM is to use standards for the interoperability between tools, the *Simulation Experiment Description Markup Language* (SED-ML) [138] is integrated into IBIOSIM to describe analysis experiments. The SBOL document, the SBML model, and the SED-ML file along with results of analysis can be collected within a COMBINE Archive [139] and uploaded to SYNBIOHUB.

Currently, the workflow in IBIOSIM requires manual design of genetic circuits. Specifically, transcriptional units are constructed by manually selecting genetic parts for building genetic circuits. In addition, this workflow requires the structure of the genetic circuits to be predefined. The work presented in this dissertation expands on the existing workflow for genetic circuit design in IBIOSIM. Rather than requiring the structure of the design to be predefined and parts to be manually selected, the workflow allows the behavior of the design to be specified and the design to be realized through automated procedures.

CHAPTER 3

ASYNCHRONOUS GENETIC CIRCUIT DESIGN

A design specification is used to describe the behavior of a circuit while ignoring its structure. Abstracting behavior from structure helps with the scalability and complexity of a design. Such an abstraction has been fundamental to the rapid development of EDA tools that are used to construct complex circuits. Verilog and VHDL are two prominent HDLs that are widely used by many EDA tools to express high-level electronic designs.

The idea of describing the behavior of a design can also be applied to GDA tools. In the proposed workflow of this dissertation, Verilog is used to describe the behavior of genetic circuit designs. While VDHL can also be used as a high-level description of genetic circuit designs, Verilog was chosen because there are existing GDA tools [15, 105] that use Verilog. By supporting Verilog in this workflow, the results produced in this work can be compared and verified with other existing tools. Furthermore, tool interoperability is critical for the growth of the GDA field since it encourages collaboration between scientists and engineers.

While describing the behavior of a design using a high-level description language helps with the reasoning about a design, the description itself is not sufficient to obtain a circuit. In order to realize a circuit, the behavioral design needs to be converted into a structural design through a synthesis procedure.

There are existing tools that allow the expression of combinational genetic circuits using HDLs [15], but there is no other tool that supports asynchronous circuit designs. This chapter describes a methodology to describe asynchronous genetic circuits using Verilog and how genetic circuits can be realized from a specification. That is, Section 3.1 goes into detail about how Verilog is used to describe behavioral designs of asynchronous circuits and Section 3.2 describes how the Verilog design is synthesized into a structural design and realized as a genetic circuit represented in SBOL.

3.1 Specification

As shown in Chapter 2, Boolean gates can be realized using biological parts through transcriptional regulation. Using such gates, genetic circuits can be constructed. Since genetic circuits have the same qualitative behavior as electronic circuits, Verilog can be used to describe the behavior of genetic circuits as well. The workflow presented in this dissertation leverages a subset of the Verilog language to express the core concepts of asynchronous circuit design. The orange labels listed under Data Conversion in **Figure 3.1** represent all the Verilog constructs that are used in this workflow.

The behavior of a genetic circuit described in Verilog is done so within a *ModuleDefinition*. Depending on the complexity of a circuit, a circuit can be abstracted and hierarchically assembled through *ModuleInstantiations* and is connected with *PortConnections*. These *PortConnections* are defined in *ModuleDefinitions* as *PortIdentifiers* to specify the direction of which signals are sending and receiving data. *Port Identifiers* have data types that can be set to *Wires* for connecting signals across different levels of a genetic circuit design or *Registers* (Reg) for storing the value of a signal. A circuit described within a *Module Definition* can have an *InitialBlock* and multiple *AlwaysBlocks*. An *InitialBlock* is used for setting the initial state of *Registers* through *BlockingAssignments*. *BlockingAssignments* execute assignments of variables in sequential order. *AlwaysBlocks* represent a grouping of Verilog constructs that are executed in a continuous loop and is used in a specification to group the behavior of a design. Within an *AlwaysBlock*, there are *BlockingAssignments*, *WaitStatements*, *ConditionalStatements*, *Delay*, and *SystemFunctions* for describing the behavior of a circuit. *WaitStatements* are used to follow the handshaking protocol for establishing a communication channel between the circuit and its stimulating environment. Unless the expression in a *WaitStatment* is evaluated to true, Verilog constructs defined after a *WaitStatement* are stalled from executing. *ConditionalStatements* are used for checking the change in signals so that the state of signal variables can be updated appropriately. *Delays* are used when the state of a variable should not be updated immediately. The value of a *Delay* can be set to a constant integer through *DecimalNumbers* or to a random value through *urandom_range*. The *urandom_range* operation is a *SystemFunction*, where a value is selected between a specified lower and upper bound. The support of *Random* is useful for randomly assigning the state of a signal that is sent to the specification. Expressions are

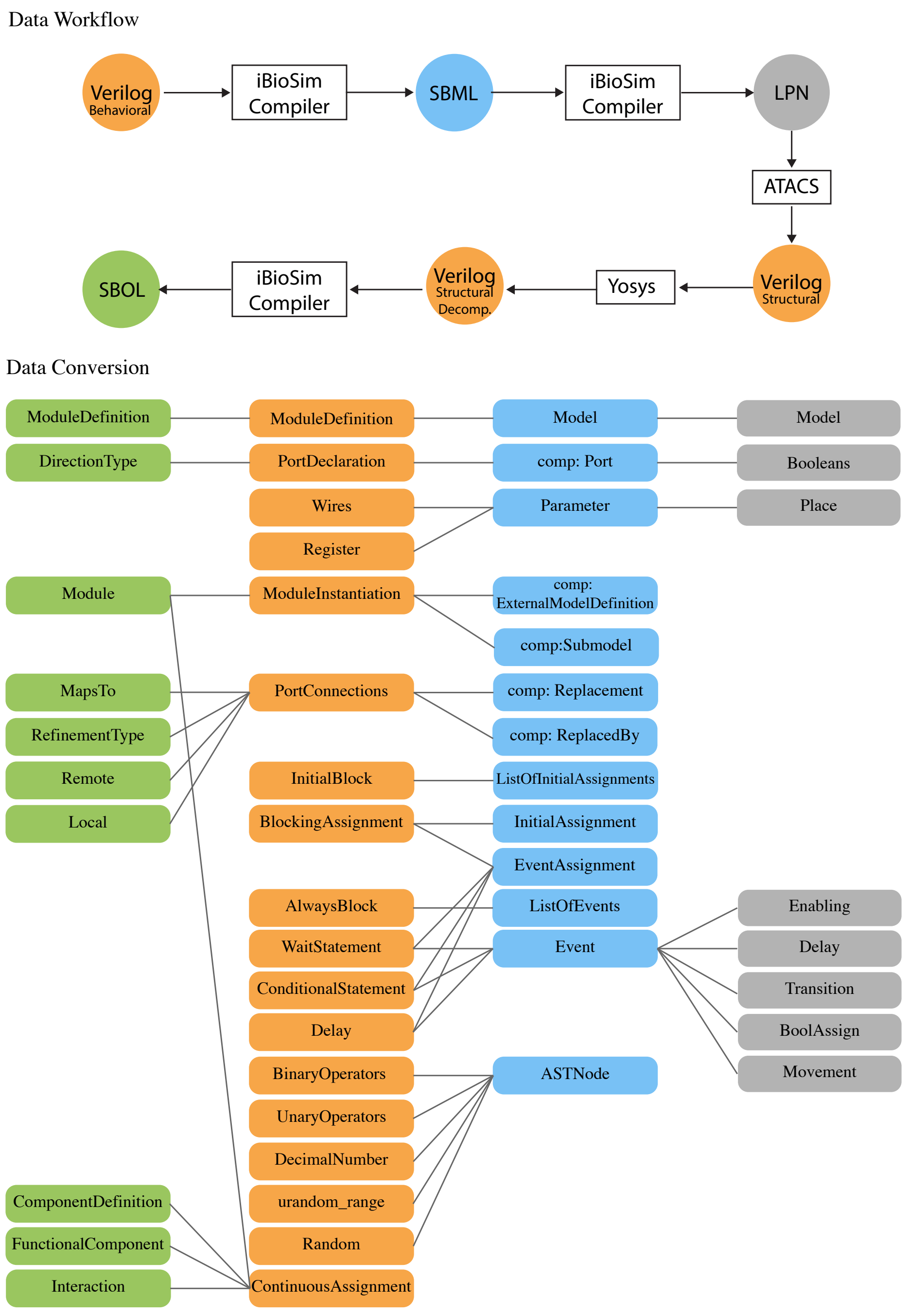


Figure 3.1: The data workflow demonstrates how a behavioral Verilog is synthesized into a structural Verilog. ATACS is an asynchronous synthesis tool used in this workflow for transforming a high-level behavioral design into a hazard-free structural design. The structural Verilog is transformed into SBOL at the end so that physical biological parts can be realized onto the structural design. A mapping of the four data formats supported in the proposed workflow is shown under Data Conversion. The green label represents the terminologies that are supported from SBOL data model, the orange labels represent the Verilog constructs that are used for designing an asynchronous genetic circuit, the blue labels represent the elements that are supported from the SBML data model, and the white label with black boarders represents terminologies that are supported within ATACS for building an LPN model.

supported with *BinaryOperators* and *UnaryOperators*. The data encodings to represent the presence of a LOW and a HIGH signal are abstracted with the support of *BinaryNumbers*.

Building upon these Verilog constructs, the S-R latch is used as an example to show how an asynchronous circuit can be represented in Verilog. **Figure 3.2** represents the specification and **Figure 3.3** represents the testbench of the circuit. The specification of the S-R latch begins with the output signals initialized to zero (LOW signal). The *AlwaysBlock* is executed and starts off by waiting on an incoming input signal. Just like a S-R latch, the combination of input signals that can be detected in the specification are when *s* is one and *r* is zero, or *s* is zero and *r* is one, or both input signals are zero. Once the input signals are set, the output signal *q* is assigned. In the case that both input signals are LOW, the output signal *q* is set to the value of the state that was last set. The current state is also updated once the output signal has been changed. This means that, if the newly assigned *q* is set to one and the previous state was set to zero, then the current state must be updated to one.

```
module srlatch_imp (s, r, q);
        output reg q;
        input wire s, r;

        initial begin
                q = 1'b0;
        end
        always begin
                wait (s == 1'b1);
                #5 q = 1'b1;
                wait (r == 1'b1);
                #5 q = 1'b0;
        end
endmodule
```

Figure 3.2: Specification of an S-R latch expressed in the Verilog language. This design style follows that of a Mealy machine in which the output signal is calculated base on the current inputs and the current state that was sensed. The *InitialBlock* is executed at the beginning to set the initial state and the output signals of the circuit. The *AlwaysBlock* is executed afterwards by first waiting on the desired input signals to become true before the circuit kickstarts and performs operations to produce an output signal. When the input signal matches what the circuit expects, then the output signal *q* is updated base on what input signals were sensed. The state of the circuit is updated after setting the output signal to reflect what the current output signal is set to in response to the input signals that were detected.

```
module srlatch_testbench ();
        wire q;
        reg s, r;
        initial begin
                s = 1'b0;
                r = 1'b0;
        end
        srlatch_imp sl_instance(
        .s(s),
        .r(r),
        .q(q)
        );
        always begin
                #5 s = 1'b1;
                wait(q == 1'b1);
                #5 s = 1'b0;
                #5 r = 1'b1;
                wait(q == 1'b0);
                #5 r = 1'b0;
        end
endmodule
```

Figure 3.3: Testbench of an S-R latch expressed in Verilog. Variations of input signals are set to test the assignment of the output signal q produced from the specification. The output signal is asserted through *WaitStatements*. If the output signal q produces in invalid value, then the *WaitStatements* is locked in that state and further input signal variations are prevented from changing.

Similarly, if the newly assigned q is set to zero and the previous state of the circuit was set to one, then the current state must be updated to zero. The last *WaitStatement* is used to ensure that the current state and the output signal q are set to the appropriate signal values before the next input signal is detected and executed.

The testbench, on the other hand, represents the testing environment for the S-R latch. The testbench begins with the input signals, s and r, initialized to zero in the *InitialBlock*. An instance of the circuit is defined after the *InitialBlock* to connect the input and output signals from the testbench to the circuit's specification. All variations of sampled input signals, assigned within the *AlwaysBlock*, are then sent to the circuit for testing the response of the circuit's behavior. First, the input signals are set so that the output signal q is set to one. Then, both input signals are set to zero so that q is expected to produce the same output value that was remembered in its previous state. In this particular condition, the output signal q is expected to stay at one. A similar behavior was tested to set the output signal q back to zero by setting input signal s to zero and r to one. Both input signals s and r are then set to zero so that the previous state of the circuit continues to hold the output signal q at zero. The wait statements found in between variations of input conditions are used to stall until q produces the expected output value before the next set of input assignments are executed. If the output signal q is not producing the desired value that the wait statement is expecting to become true, then this error can easily be recognized during simulation when the output value remains in the same state and prevents further input variations to change.

3.2 Synthesis

Labeled-petri net (LPN) models are widely used for specifying and synthesizing asynchronous circuits. The LPN models, as shown in **Figure 3.4**, are a directed bipartite graph. An LPN model has *Places* to represent statement *Assignments*. Each *Place* can consume a *Token* and a *Token* can have a *Marking* to indicate an enabling of the *Assignment*. A *Token* can move from a *Preset* to *Postset* through *Transitions*. *Preset* is defined as an input *Place* to a *Transition* and a *Postset* is defined as an output *Place* of a *Transition*. A *Transition* represents an action that can fire when the *Condition* of a *Transition* is satisfied and the *Markings* on the *Presets* are enabled. A *Delay* can be added to a *Transition* to control the time for which a *Transition* can fire. When a *Transition* fires, the token is removed from the *Presets* and a *Marking* is made on the *Postsets*.

The workflow presented in this dissertation leverages the ATACS tool for synthesis. ATACS does not support Verilog files as input. Therefore, one contribution of this dissertation is the implementation of a Verilog compiler that can convert a behavioral Verilog file to an LPN model so that synthesis can be performed in ATACS. **Figure 3.1** shows a data workflow of how this is done. When a behavioral Verilog file is provided, it is first translated into an SBML model and then to an LPN model that ATACS can parse to perform synthesis. SBML is used as the intermediate data model when converting from behavioral Verilog to an LPN model for two reasons. First, the format for describing LPN models that is used in ATACS does not support hierarchy and many designs described in Verilog are hierarchical. However, this issue can be avoided by making use of the SBML

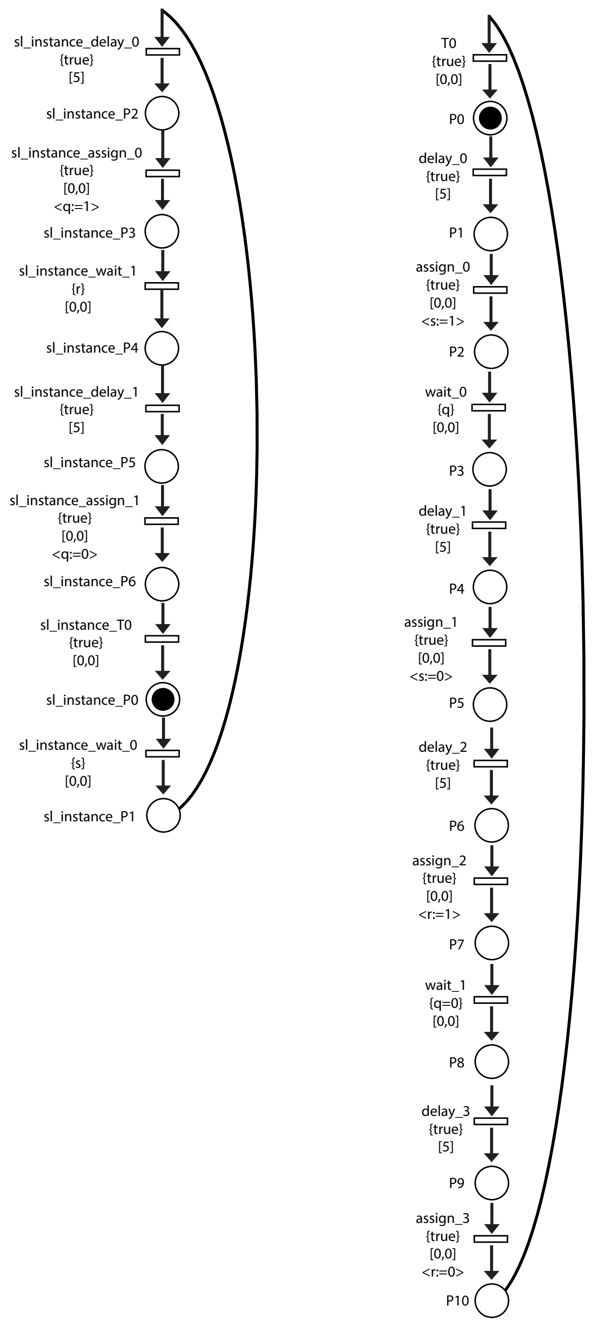


Figure 3.4: An example of the SR latch LPN models generated from ATACS. The left LPN model represents the specification transformed from **Figure 3.2** and the right LPN model represents the testbench transformed from **Figure 3.3**. The circles are *Places* to represent conditions. The black filled circle is a *Token* with a *Marking* to represent that condition of the place holds true. The *Marking* of these *Tokens* can move from one place to a subsequent place base on the horizontal bars that are referred to as *Transitions*. These *Transitions* represent actions that can fire when the places connected towards the transition have conditions that hold true.

data model. The SBML data have many core elements that can encode data models that follow the LPN paradigm. SBML has the comp extension package that can be used for expressing hierarchical models and these hierarchical designs can be flattened using the SBML libraries [140, 141]

Each color label shown under Data Conversion for **Figure 3.1** shows a list of terminologies that are encoded into its corresponding data model under the Data Workflow. A mapping of terminologies going from one data model to another form is shown with lines that connect the terminologies together. Using the S-R latch as an example, the *ModuleDefinition* for the specification and the testbench are converted into an SBML *Model*. The *ModuleInstantiation* is used for instantiating the specification onto the testbench, and it is converted into *ExternalModelDefinition* and *Submodel* in SBML. *ExternalModelDefinition* is used for referencing the SBML file of the specification and a *Submodel* for instantiating onto the SBML model that defines the testbench. The *PortIdentifiers* are converted into SBML *Ports* and are set with Sequence Biology Ontology (SBO) [142] terms for further distinguishing the input and output signals. *PortConnections* are converted into *Replacements* and *ReplacedBys* for mapping signals across hierarchical models. In this example, the input signals for *s* and *r* in the testbench are set to *Replacement* so that the input signals from the testbench are used to assign the input signals within the specification and the output signal *q* in the testbench is set to a *ReplacedBy* so that the output signal in the specification replaces the output signal found in the testbench. *BlockingAssignments* in the *InitialBlocks* are converted into *InitialAssignments*. The *AlwaysBlocks* are converted into a *ListOfEvents* that compose a LPN. Each *Event* corresponds to a transition, and the first one has the *Preset* places with a marking. *WaitStatements*, *ConditionalStatements*, and *Delays* found within an *AlwaysBlock* are converted into *Events* as well where the *Postset* of the last statement is the *Preset* of the first statement within the *AlwaysBlock*. The Verilog expressions are converted into an *ASTNode* that can be assigned onto each *Event*.

First, LPN models do not not support hierarchical models. In order to do so, a flattening routine must be implemented from Verilog to LPN. However, this issue can be avoided by making use of the SBML data model. SBML have many core elements that can encode data from a structural Verilog to an LPN model. SBML has the comp extension package that can be used for expressing hierarchical models that are designed in Verilog. SBML also has flattening methods that can be used for modeling information into an LPN model. Second, expressing a genetic design in SBML allows for the specification to be simulated within iBioSim before and after performing synthesis. Synthesis can report failure or pass messages and having an SBML model for the specification can allow for easy debugging in the case that synthesis fails. SBML can also be used for further verification to be performed at different design stages when synthesis passes. More details of this verification process are mentioned in Chapter 6.

3.2.1 Behavioral Verilog to SBML

The S-R latch designed in Section 3.1 is used as an example to illustrate the compilation of Verilog files into an LPN model. The *Modules* for the specification and testbench are converted into SBML *Models*. The *PortIdentifiers* for signals *s*, *r*, and *q* are converted into SBML *Ports* and are set with *Sequence Biology Ontology* (SBO) terms for further distinguishing input and output signals. *Wires* and *Registers* are converted into *Parameters*. A *ModuleInstantiation* that references the specification into the testbench is converted into *ExternalModelDefinition* and *Submodel*. *ExternalModelDefinition* is added into the SBML testbench file so that it can refer to the SBML specification file. *Submodel* is used for instantiating a copy of the specification into the SBML testbench *Model*. *PortConnections* are converted into *Replacements* and *ReplacedBys* for mapping signals from the testbench to the specification. In this example, two *Replacements* objects are created for *s* and *r* so that the input signals from the testbench replace the signals within the specification. A *ReplacedBy* is created for the output signal *q* so that the signal in the specification replaces the output signal found in the testbench. *InitialBlocks* represent a *ListOfInitialAssignments* and each construct nested inside this block is converted into an *InitialAssignment*. *AlwaysBlocks* represent a *ListOfEvents*. *Events* have their own *Parmeter* variable and they are created from *BlockingAssignments*, *WaitStatements*, *ConditionalStatements*, and *Delays* nested within an *AlwaysBlock*. An *ASTNode* is added onto a *Trigger* of an *Event* for representing the expression of a Verilog construct. This *ASTNode* supports the expression of *BinaryOperators*, *UnaryOperators*, *DecimalNumbers*, *urandom_range*, *Random*, and *ContinuousAssignments* in Verilog constructs. The order of Verilog constructs defined within an *AlwaysBlock* is maintained by setting the value of *Parameters* found on *Triggers* and *EventAssignments*.

When the expression of a *Trigger* becomes true, then the *Parameter* variable assigned to its *Event* is enabled. Enabling of a *Trigger* on an *Event* causes the *EventAssignment* to change the value of the *Parameter* variable assigned to this *Event* to a *Paremeter* assigned to a different *Event*. Since the *AlwaysBlock* operates as a continuous loop for executing all nested Verilog constructs within its block, the first *Event* created is initialized from its *Parameter* variable to indicate the starting position of a continuous loop.

3.2.2 SBML to LPN

As shown in **Figure 3.1**, the next step in the data workflow after Verilog is converted into SBML models is to create an LPN model. To do so, these SBML *Models* are flattened by calling iBioSim's SBML flattening method. The result of the flattener produces an SBML model composed of core elements that describe both the specification and the testbench. Elements of an LPN model are created by using the LPN data model supported in iBioSim.

The flattened SBML *Model* is converted into an LPN *Model*. *Parameters* that are not categorized as primary inputs, outputs, and registers are converted into *Places*. Each *Event* is converted into a *Transition*. The *EventAssignments* are added as *Assignments* to the created *Transition*. If the variable of the *EventAssignment* refers to the variable of a *Place*, a *Movement* is created to connect the *Place* to the *Transition*. In the case that no *Place* matches the variable name of an *EventAssignment*, a *BoolAssign* is created. If the *EventAssigment* is encoded from a Verilog *Random* construct, this *Event* is dropped. SBML *Ports* are converted into *Booleans* for tracking input signals, output signals, and internal states of the circuit. *Triggers* are converted into *Enablings*. *Delays* that are set in an *Event* are converted into equivalent LPN *Delays*.

3.2.3 Synthesizing in ATACS

The LPN model derived from SBML is sent to ATACS to create a state graph needed for performing synthesis. The state graph is analyzed to determine the logic necessary to implement any state and output signals. After the state graph is generated, it is checked to see if it satisfies the Complete State Coding (CSC) property.

CSC violations ensure that there are no confusion on which current and next state is receiving and sending data.

A key challenge with asynchronous logic design is the avoidance of *logic hazards*. A logic hazard occurs if a signal that is supposed to remain stable momentarily changes value, or a signal that is supposed to change does so non-monotonically [143]. Unlike electronic circuit designs where the cost of errors is circuit failure, in genetic circuits, errors can be tolerated. Since genetic circuits are deployed in a population of cells, there are multiple copies of a genetic circuit available. Therefore, if one genetic circuit fails within a cell due to glitches, it may not highly affect the entire system if other cells exhibit the correct behavior. Nikolaev et al. [144] have shown from modeling a genetic toggle switch that a large population of cells performing the same task will help average out the expected behavior for the majority of the genetic toggle switch circuits. While it is important to address high probability hazards, genetic circuits can likely tolerate low probability hazards. Exploiting this observation is an area of future research that we plan to explore.

The Verilog constructs that are mentioned thus far are used for describing the behavioral design of a genetic circuit. A structural representation of a genetic circuit is much more simplified in which a *ModuleDefinition* does not have *InitialBlocks* and *AlwaysBlocks*. Instead, a structural representation for describing a circuit is done so through *ContinuousAssignments*. A *ContinuousAssignment* is dependent upon the expression of an assign statement becoming true in order for the variable of the assignment to be updated with a new output value for an output signal or an internal state variable. Synthesis produces a structural Verilog design from a behavioral Verilog representation. This representation contains continuous assign statements expressed in the form shown in **Figure 3.5**.

```
module r_s_q_net(r, s, q);

  input r;
  input s;
  output q;

assign q = (s) | (~r) & q;

endmodule
```

Figure 3.5: A synthesized design of the S-R latch expressed in structural Verilog made of *ContinuousAssignments*. The variable of a *ContinuousAssignment* updates it value when its expression becomes true. This structural Verilog is converted into a decompose form represented in the SBOL data format for describing the genetic circuit.

In the case that sequential circuits are designed using this workflow, ATACS produces expressions that include feedback to store state. These expressions in this structural Verilog file can be further simplified in another EDA synthesis tool called YOSYS so that the number of logic gates needed to represent this genetic circuit is reduced. After simplification is performed, YOSYS can produce a Verilog file that is similar to ATACS that is made of *ContinuousAssignments*. These structural Verilog files are then decomposed into the SBOL data format with only *NOT* and *NOR* logic functions to describe the structural design of a genetic circuit.

3.2.4 Structural Verilog to SBOL

An SBOL *ModuleDefinition* is created from a Verilog *ModuleDefinition* for representing the genetic circuit of an S-R latch. The input signals and output signals are converted into *ComponentDefinitions* for representing proteins. *DirectionType* is assigned to a *ComponentDefinition* to differentiate between input signals, output signals, or a register variable for holding state. Conceptually, a *ModuleInstantiation* is converted into a SBOL *Module*. Because a structural Verilog does not use *ModuleInstantiation*, hierarchical designs of a genetic circuit can be built through each *ContinuousAssignment*. *MapsTo* objects are used to connect variables of a *ContinuousAssignment* across different *ModuleDefinitions*. A *RefinementType* is set to a *MapsTo* for indicating which *FunctionalComponent* should be used in hierarchical designs. A *RefinementType* can be set to *Local* so that it can refer to a *FunctionalComponent* within a logic gate signal or it can be set to *Remote* for referring to a *FunctionalComponent* on the top level circuit where the entire circuit is formed. The expression of a *ContinuousAssignment* is parsed into an *ASTNode* to perform decomposition.

Decomposition involves transforming Boolean logic functions to express a design using *NOT* and *NOR* logic gates. This is accomplished by performing DeMorgan's Theorem on each expression. Because synthesis produces *ContinuousAssignments* expressed in the Sum-of-Products (SOP) form, the decomposition method supports expressions with *NOT*, *AND*, and *OR* operators.

Setting the expression of a *ContinuousAssignment* into an *ASTNode* allows the decomposition method to easily access different operators that are found within the given expression. This means that each operator is broken into an individual *ASTNode* with the operands set as children. These individual *ASTNodes* are then layered onto each other to form the complete expression. The layered *ASTNodes* are iterated from left to right by parsing the innermost expression, which works its way out to create a decomposed expression. It is important to note that while the decomposition method can decompose an expression to use more than two input logic gates, this work limits the decomposition to two input logic gates because of genetic constraints. These genetic constraints are discussed in further detail in Chapter 4.

In the example shown on **Figure 3.5**, the expression for assignment q is represented in the following format as an *ASTNode*: $or(s, and(not(r), q))$. Working from left to right, not(r) is the innermost expression that is handled. Because $not(r)$ is already in a decomposed form, this *ASTNode* is replicated into the decomposed expression. Next the *ASTNode* for the AND operator is handled. An AND logic is decomposed into a NOR gate with its operands inverted. This decomposed AND logic is added onto the previous decomposed expression to form the following: $nor(not(not(r)), not(q))$. Finally, the OR logic is handled by decomposing the expression with two inversions to form *not(nor(s, nor(not(not(r)), not(q))))*. This final decomposed expression is returned from the decomposition method to construct a genetic circuit.

The concept that was described in Chapter 2 for building a genetic NOT gate and a genetic NOR gate out of transcriptional regulation is applied in this part of the conversion to build the decomposed logic into a genetic circuit. The operands of a logic function are converted into *FunctionalComponents* for representing transcriptional units and proteins. The operators of a logic function are converted into *Interactions* for representing the relationship between proteins to and from transcriptional units.

3.3 Summary

In synthetic biology, large circuits have not been realized yet. Biology is far too complex, and in order to deal with such complexity, abstraction is critical. Boolean abstraction is one such abstraction that has been applied to synthetic biology. That is, inspired by the success of digital circuits, genetic circuits can be described in terms of LOW and HIGH values.

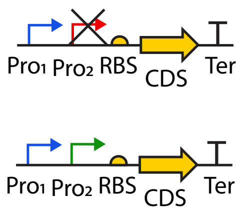
Since genetic circuits can be abstracted as electronic circuits, some methodologies used to design electronic circuits can also be used to design genetic circuits. For instance, genetic circuits can be designed using HDLs that describe the design's behavior rather than structure. This chapter demonstrates how asynchronous circuit designs described using Verilog can be realized as genetic circuits described in SBOL.

CHAPTER 4

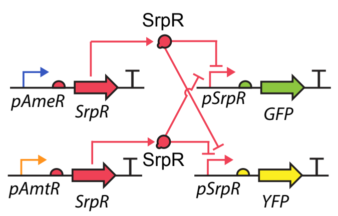
PROGRAMMATIC CREATION OF GATE LIBRARIES THROUGH AUTOMATED GATE GENERATION

CAD tools are essential to deal with the complexity of genetic circuit design. An important element of CAD tools used for genetic circuit design is the access to collections of well-characterized parts used to realize a design. The number of well-characterized parts influence the scale of the design. That is, the larger the number of parts in the collection, the larger the design that can be realized. However, this poses a key challenge: how to efficiently explore the solution space of possible designs to find the best possible design. A solution space is a collection of valid designs, where a valid design is one that does not violate any design constraint. In genetic circuit design, design constraints include *crosstalk*, *signal mismatch*, *roadblocking*, and *genetic context effects* [145]. As shown in **Figure 4.1a**, roadblocking can occur in gates that use tandem promoters [146], and the downstream promoter interferes in transcription initiated at the upstream promoter [15]. As shown in **Figure 4.1b**, crosstalk occurs in genetic circuits when the product of one gate has unintended interactions with another. As shown in **Figure 4.1c**, a signal mismatch occurs when the level of a product produced by one gate does not meet the threshold necessary to cause the correct response in a downstream gate. Finally, as shown in **Figure 4.1d**, genetic context effects occur when the ordering of neighboring components changes the behavior of the design. In cases where context effects are known to be strong, gates may be integrated into the host genome at predetermined locations where each gate's transcription rate has already been characterized.

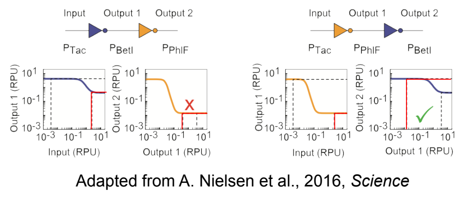
Genetic circuit design constraints reduce the solution space of genetic circuit designs. For example, avoiding crosstalk requires that a particular signal carrier (transcription factor) assigns to a specific node in the circuit and does not interact with other nodes within



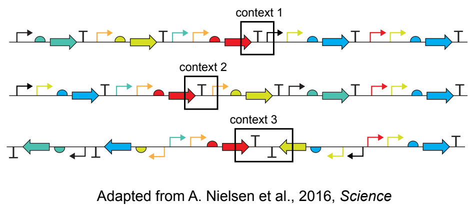
(a) Roadblock



(b) Crosstalk



(c) Signal Mismatch



(d) Genetic Context Effects

Figure 4.1: Four genetic constraints that occur in genetic circuits. **Figure 4.1a** Roadblock: initiation of transcription from the upstream promoter in a tandem promoter is impeded by the presence of a transcription factor bound to the downstream promoter. **Figure 4.1b** Crosstalk: interference of circuit components with each other or the host circuitry. **Figure 4.1c** Signal mismatch: incompatible signal levels of gates composed in series. **Figure 4.1d** Genetic context effects: the same circuit can act differently based on the ordering of neighboring components.

the circuit. Such a constraint can be avoided by having a diverse set of signal carriers. Signal mismatching can be avoided by having a diverse set of gates with different input and output threshold levels. Roadblocking is avoided by finding promoter interferences during the gate library characterization and eliminating these problematic permutations from the ensemble of DNA sequences that are considered to fulfill the specification. Therefore, it is critical to have a library with diverse biological parts.

Some existing GDA tools are using libraries with characterized parts to implement a genetic circuit design [64, 105]. There are also existing GDA tools with customized libraries with tool-specific usage [15, 103, 104]. Because there are various ways of building genetic logic gates with similar logic behavior, the construction of these gate varies across different tools. The main contribution of this chapter is to present a standardized framework for the creation of a gate library through automated gate generation. Specifically, Section 4.1 goes over the process of assembling biological parts to form transcriptional units using SynBioHub and SBOLDesigner. Section 4.2 describes the process of automating the creation of genetic logic gates using VPR model generation supported within iBioSim. Section 4.3 goes over the process of identifying genetic gates into their proper gate types.

Then, Section 4.4 concludes by summarizing the work presented in this chapter.

4.1 Assembling Transcriptional Units

The first step when building a library of genetic logic gates is to generate a large number of transcriptional units so that interactions can be added to connect proteins, small molecules, and complexes as the input and output signals to a genetic logic gate. SBOLDESIGNER [102] is a CAD tool that was chosen for this task because it is supported in iBIOSIM as a plugin for assembling DNA parts to build a DNA-level design. This CAD tool has the ability to connect to the online design repository SYNBIOHUB [147] to retrieve linked information for genetic parts and build large quantities of transcriptional units using *CombinatorialDerivation* [148]. A *CombinatorialDerivation*, as shown in **Figure 4.2**, has three property fields that must be set in order to build transcriptional units in volume. The first property is a *Template* that refers to a *ComponentDefinition* for representing a transcriptional unit. The second property is a *VariableComponent* used for listing possible *Variants* of DNA parts assigned to a component on a transcriptional unit. The last property is a *Strategy* used to indicate how a list of *Variants* can be *Enumerated* or *Sampled* on the *Template*. *Enumerated* indicates that all possible outcomes of DNA parts listed in each *VariableComponent* are used to create different variations of transcriptional units. *Sampled* indicates a subset of DNA parts are selected from each *VariableComponents* to form different variations of transcriptional units. The number of DNA parts selected from each *VariableComponent* for building the template is set using the *Operator* field.

Figure 4.2 shows an example of a *CombinatorialDerivation* created for building transcription units with one or two promoters. The *Template* references a transcriptional unit made of tandem promoters, an RBS, a CDS, and a terminator. Each component on the transcriptional unit has a *VariableComponent* created for listing the possible DNA parts that are chosen when assembling multiple transcriptional units. In this specific example, three *VariableComponents* are created and are referred to in **Figure 4.2** as *vc*1, *vc*2, and *vc*3. The DNA parts are referred to as *Variants*, and they are added to a *VariableComponent* in SBOLDESIGNER by fetching this information from a *Collection* in a SYNBIOHUB repository. The number of DNA parts that are selected from each *VariableComponent* when building a transcriptional unit is specified by the *Operator* (op). The *Operator* for *vc*1 and *vc*3 is

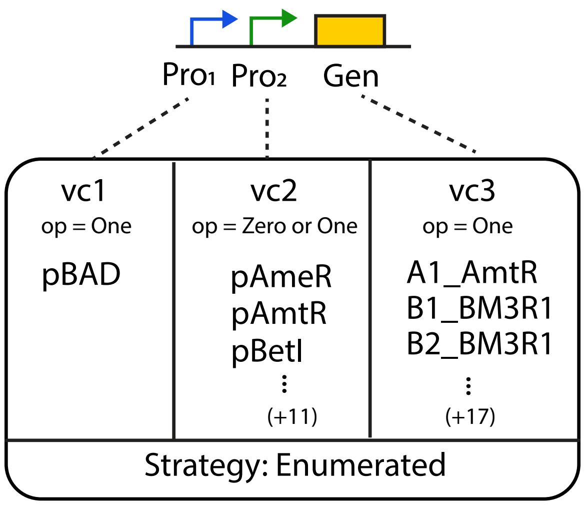


Figure 4.2: An illustration of the properties that must be set on a *CombinatorialDerivation* for generating transcriptional units to build a preset of genetic logic gates supported in this workflow. The pattern for building multiple transcriptional units with two promoters, two pairs of RBS, CDS, and a terminator is specified as the *Template*. Three *VariableComponents* (*vc*1, *vc*2, and *vc*3) are created for listing possible DNA parts that a component on the transcriptional unit can be assembled from. The DNA parts listed in these three *Variable-Components* are referred to as *Variants*. *VariableComponents* for *vc*1 and *vc*3 have *Operators* (*op*) set to one to indicate that one *Variant* from its list must be selected when constructing a transcriptional unit. Likewise, *VariableComponent vc*2 has *op* set to *zeroOrOne* to indicate that zero or one *Variant* can be selected from its list to generate transcriptional units with tandem promoters and gates with an output reporter included. The *Strategy* is set to *Enumerated* so that all *Variants* listed in each *VariableComponent* must be used to build different variations of a transcriptional unit with one and two input gates.

set to *One* and *v*2 is set to *ZeroOrOne* because the main requirement for this *CombinatorialDerivation* is to generate transcriptional units that must have at least one promoter, one RBS, one CDS, and one terminator so that one input genetic logic gate can be formed. For the downstream promoter with *op* set to *ZeroOrOne*, this guarantees that the transcriptional unit provided as the *Template* can at most assemble two promoters to form a transcriptional unit that can be generated into two input genetic logic gates. Lastly, the *Strategy* of this *CombinatorialDerivation* is set to *Enumerated* so that all DNA parts listed in each *VariableComponents* are used to build different combinations of transcriptional units specified by the *Template*.

Roadblocking is resolved by handpicking compatible promoters that are added as *Variants* to its *VariableComponent*. However, this can only happen when the dataset used for building transcriptional units has detailed information on which pair of promoters pair well with each other.

The result of *CombinatorialDerivaton* is returned in SBOLDESIGNER as a new *SBOLDocument*. Each transcriptional unit listed in this new SBOLDocument is represented as a *RootComponentDefintion*. The number of transcriptional units that are generated from this *CombinatorialDerivation* is calcuated by multiplying all *Variants* from each *VariableComponent*. For the example that is shown in **Figure 4.2**, 300 transcriptional units are expected to be generated from the *CombinatorialDerivation* with 20 transcriptional units that are made of a single promoter and 280 transcriptional units that are made of tandem promoters.

4.2 Gate Generation

Once transcriptional units are created, the next step involves transforming them into genetic logic gates using VPR [149]. VPR facilitates model composition by mining for biological components and functional relationships from SYNBIOHUB instances that could be used to optimize an existing DNA-level design or to create a new design. It is important to note that if DNA parts are taken from one dataset on SYNBIOHUB for assembling transcriptional units and a different dataset is used when calling VPR, there is a possibility that VPR is unable to find data to enrich a given transcriptional unit.

Each transcriptional unit must be transferred into a separate *SBOLDocument* before calling VPR to ensure that VPR does not combine unintended *Interactions* between multiple transcriptional units when building a genetic logic gate. This process is performed by recursively copying each *RootComponentDefinition* to its own *SBOLDocument*. VPR is then called on each *SBOLDocument* so that the design can be enriched with additional SBOL data.

The VPR API creates an SBOL *ModuleDefinition* for the DNA-level design. Then, it adds to the *ModuleDefinition* proteins, small molecule, and complex that interact with the components in the design. In the example shown in **Figure 4.3**, four *Interactions* are found and added to the design. These four *Interactions* are protein X inhibiting Pro, Protein Y produced from CDS, and two degradation interactions for X and Y. Black arrows are *MapsTo* objects used to connect the proteins and DNA parts from the *Interaction Modules* to the transcriptional unit. VPR returns this information as a new *SBOLDocument* that can then be used for sorting gate types.

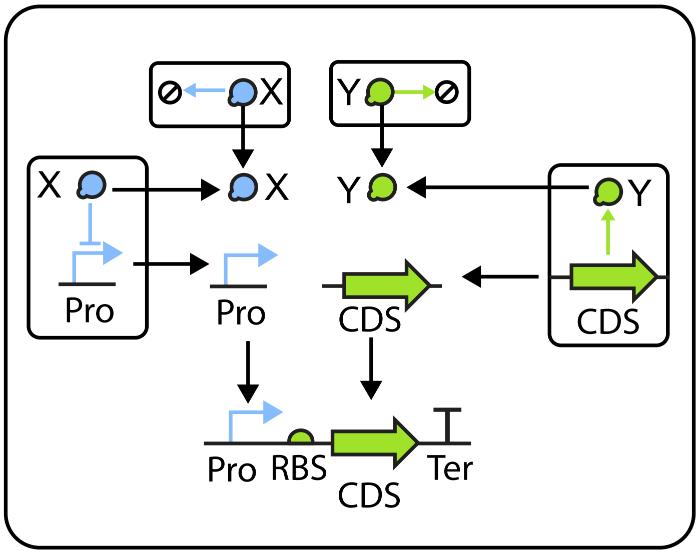


Figure 4.3: An enriched design of a transcriptional unit after calling VPR. Four *Interactions* encapsulated within *Modules* are added to the design. These four *Interactions* are an inhibiting interaction for protein *X* and *Pro*, a production interaction for protein *Y* and CDS, and two degradation interactions for protein *X* and *Y*. The black arrows denote *MapsTo* objects that are used for connecting proteins and DNA parts from the *Modules* encasing the four types of *Interactions* to the components on the transcriptional unit.

4.3 Identifying Gate Types

The enriched designs generated by VPR goes through a gate identification procedure to group the different structures of a genetic logic gate that exhibit the same logic behavior. The algorithm for this process is described in **Algorithm 4.1**. This method takes a *ModuleDefinition*, *gateMD*, generated from VPR and produces an identified genetic gate that has been classified as a NOT, NOR, OR, AND, or $NAND$ gate. This method begins by creating two list of components, p and cds, for storing the promoters and CDS that the gate has on its transcriptional unit. A map of components, σ, is also created to store the molecules and the *Interactions* that connect them to the transcriptional unit. Then, molecules are collected by iterating through *FunctionalComponents* from *gateMD*. Each *FunctionalComponent*, fc, is added into c. Recall from **Figure 4.3** that *MapsTo* object are used extensively in the enriched genetic gate design produced by VPR. Storing all fc instances used in *gateMD* ensures that the local parts that the *MapsTo* object referred to in the *gateMD* are accessible later on when parts and molecule information are needed to identify the gate. Because the promoter and the CDS region is defined in this workflow as the region where input molecules and output proteins interact directly to the transcriptional

Algorithm 4.1: GateIdentifier

```
Inputs: ModuleDefinition $gateMD$;
Output: Genetic Gate $g$;
$p = \langle\rangle$;
$cds = \langle\rangle$;
$\sigma = \langle\rangle$;
foreach $fc$ in $gateMD.FunctionalComponents$ do
    component = fc;
    insert($\sigma, component$);
    $cd = fc.getCd()$;
    if $cd$ is PROMOTER then
        add($p, component$);
    else if $cd$ is CDS then
        add($cds, component$);
foreach $m$ in $gateMD.getModules$ do
    addGateInteraction($m, \sigma$);
return getGateType($p, cds$);
```

unit, the fc for these DNA parts are also stored in p and c. The lists p and c are used afterwards when all information has been collected about the genetic gate.

The second step involves collecting the type of *Interaction* and its *Participants* that are involved in connecting components together to form the logic behavior of a genetic gate. This step is done by iterating through all *Modules*, m, instantiated in *gateMD*. Then **addGateInteraction** shown in **Algorithm 4.2** is performed for each m by linking interactions to the components collected in σ.

The first step in **Algorithm 4.2** is to collect *FunctionalComponents* that are found in *ModuleDefinition* m. Then, m is stored in $refMD$ so that the information about *Interactions* is retrieved. Each *Interaction*, i, found in $refMd$ is recorded by first collecting information about its *Participants*. The lists $inputs$ and $outputs$ are created to store when *Participant*, p, acts as inputs and outputs to i. In this case, the role of p is used as a reference to track components that act as the input to an *Interaction* and which component represents the output from an *Interaction*. If the role of p is *INHIBITOR*, *STIMULATOR*, *TEMPLATE*, or *REACTANT*, then p is categorized as an input to i. Likewise, if the role of p is *INHIBITED*, *STIMULATED*, or *PRODUCT*, then p is categorized as an output to i. The $components$ added to $inputs$ and $outputs$ are retrieved by first getting the *FunctionalComponent* referred to in $refMd$ and storing it into $localPart$. Then, ω is used to retrieve the $gatePart$ that

Algorithm 4.2: addGateInteraction

```
Inputs: Module $m$, Map of Components $\sigma$;
$\omega$ = getMapsToParts(module.MapsTos);
refMd = $m.MD$;
foreach $i$ in $refMd.Interactions$ do
    inputs = $\langle\rangle$;
    outputs = $\langle\rangle$;
    foreach $p$ in $i.Participations$ do
        localPart = $p.Participant$;
        gatePart = getPartMappings($\omega, localPart$);
        component = getComponent($\sigma, gatePart$);
        if $getRole(gatePart, INHIBITED) \lor getRole(gatePart, STIMULATED) \lor getRole(gatePart, PRODUCT)$ then
            add(outputs, component);
        else if $getRole(gatePart, INHIBITOR) \lor getRole(gatePart, STIMULATOR) \lor getRole(gatePart, TEMPLATE) \lor getRole(gatePart, REACTANT)$ then
            add(inputs, component);
    iType = getInteractionType(i);
    if iType != UNKNOWN then
        foreach out in outputs do
            addOutputInteraction(iType, inputs);
        foreach in in inputs do
            addInputInteraction(iType, outputs);
```

connects to *localPart* from *gateMD*. The *gatePart* is then used to look up the component from ω to add onto the list of *inputs* and *outputs*. At this point in **Algorithm 4.2**, all *inputs* and *outputs* for *i* have been collected. The next step is to retrieve the interaction type for *i* and storing it into *iType* by calling **getInteractionType**. As long as *iType* is not classified as *UNKNOWN*, then the components stored on the list of *inputs* and *outputs* are linked together with this *iType*. This method continues until all *Interactions*, *i*, have been traversed.

Once all information about the gates has been collected, then the last step in **Algorithm 4.1** is to determine the gate type by calling **getGateType**. This method takes the list for *p* and *cds* and produces a genetic gate, *g*, that has been identified into its proper type. This method traverses all *components* that are linked to each promoter in *p* and coding sequences in *CDS* to identify the gates that are shown in **Figure 4.4**. If there is an interaction inhibiting a promoter, *Pro*, and an interaction forming a production from a CDS to a protein, *Y*, then the gate is identified as a *NOT* gate. *OR* gates are identified by looking for a transcriptional unit with two transcription factors, X_0 and X_1 activating a

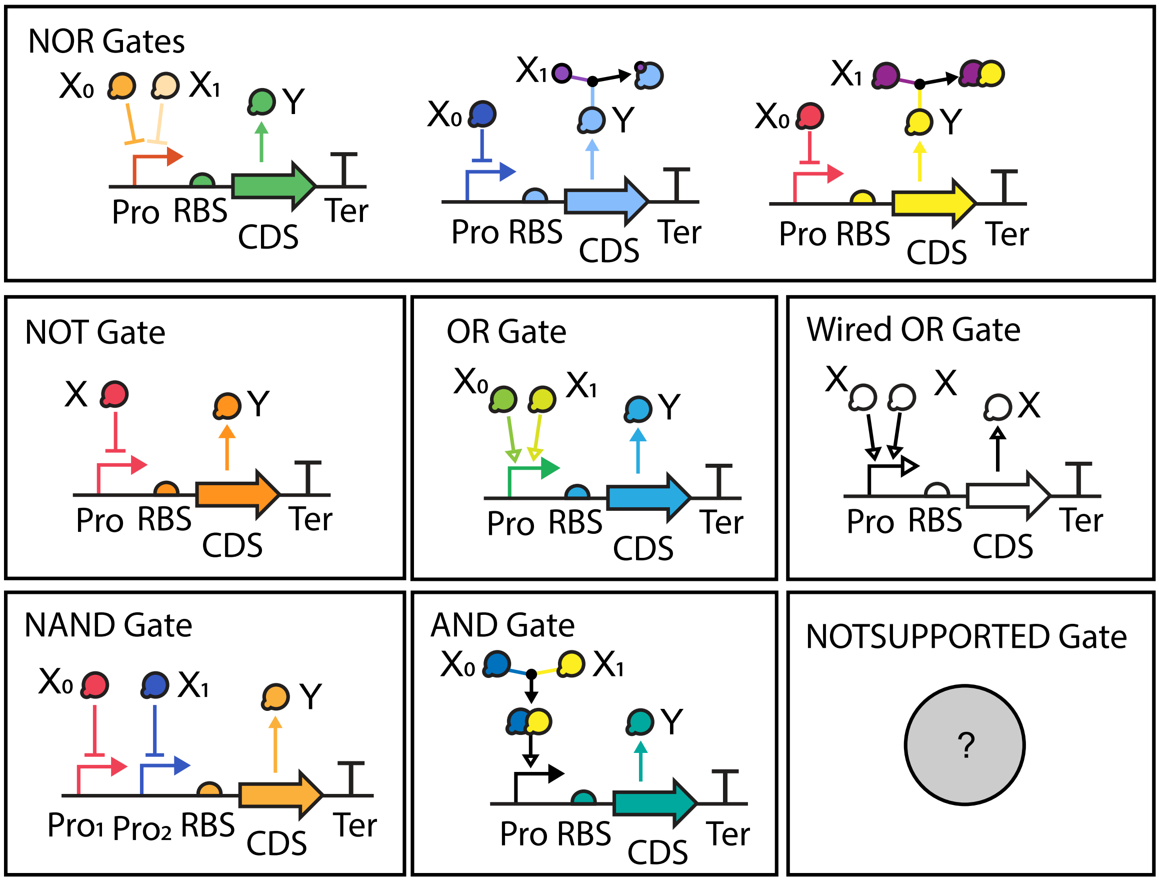


Figure 4.4: Genetic logic gates that are identified and sorted in the gate generation process are *NOT*, *NOR*, *OR*, wired *OR*, *NAND*, *AND*, and *NOTSUPPORTED* gates. A *NOT* gate is identified as a transcriptional unit with a promoter repressed by an input protein X that produces an output protein Y. An *OR* gate is identified as a transcriptional unit with a promoter activated by two input proteins X_0 and X_1 that produces an output protein Y. A wired *OR* gate has identical structure as that of an *OR* gate but the input and output signals, in this case, are all identical. A *NAND* gate is identified as a transcriptional unit with tandem promoters that are both separately inhibited by two input proteins X_0 and X_1 that produces an output protein Y. A *AND* gate is identified as two input proteins X_0 and X_1 forming a protein-to-protein complex to repress Pro. When the protein-to-protein complex is not present, the output protein Y is produced. Because there are several ways to build a genetic logic gate that exhibit the same logic behavior, three different structures for a genetic *NOR* gates are supported. The first structure has a transcriptional unit with two transcription factors repressing a promoter (*Pro*) and producing an output protein Y. The second structure has two forms of input molecules. The first is an input protein X_0 repressing the *Pro* and the second input is a small molecule X_1 forming a complex with the output protein Y. The third structure ressembles the second structure in which the second input signal X_1 forms a complex with the output signal. The difference in the third structure in comparison to the second structure is that X_1 is an input protein rather than a small molecule. Gates that are not identified as a *NOT*, *OR*, wired *OR*, *NAND*, *AND*, and *NOR* gates are classified as *NOTSUPPORTED* Gates.

promoter *Pro* and a protein Y produced from a CDS. *NAND* gates are identified with a transcriptional unit with two transcription factors X_0 and X_1 repressing tandem promoters and produce an output protein Y from the CDS region. *AND* gates are identified as two proteins, X_0 and X_1, forming a complex to activate a promoter, *Pro*, and produce an output protein, Y.

NOR gates are identified for two type of structures. The first structure of a genetic *NOR* gate is made of two transcription factors X_0, and X_1, repressing one common promoter with an output protein Y produced from the CDS region. The second structure defines the input of X_1 as a small molecule or a protein that forms a complex with the output protein Y. In this specific gate structure, the molecule signal for X_1 can go HIGH when the output protein Y is present. There is one special condition, however, when identifying the second structure *NOR* gate. Since this second structure comes from a transcriptional unit that was designed from half of a genetic toggle switch, extra information is included in this *NOR* gate after calling VPR. This extra information is shown in **Figure 4.5** as the small molecule X_2 forming a complex with X_0. The gate identifier allows the identification of such gates without removing information. As long as this type of *NOR* gate fits the primary description that is expected in the second structure, this type of gate is recognized in **Algorithm 4.1** as a *NOR* gate.

Wired *OR* gates are also supported in the gate generation procedure. Wired *OR* gates

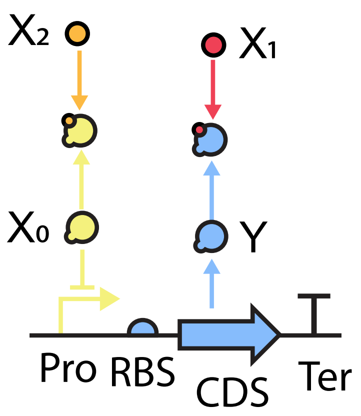


Figure 4.5: Extra information included from VPR when identifying *NOR* gates. This type of gate is expected to be identified as the second structure *NOR* gate shown in **Figure 4.4**. Important information might be lost by removing this extra information that is added from VPR. As a result, this type of *NOR* gate is recognized in the gate identify step unchanged as long as this *NOR* gate has the same substructure as the expected *NOR* gate.

are advantageous when the types of gates available to use in the workflow are limited. For example, if the technology mapping procedure needs *NAND* gates but none are available, then wired *OR* gates can be used to build such a gate.

NAND gates built from wired *OR* gates are made by connecting two *NOT* gates that produce the same output protein. The wired *OR* gate takes in two input proteins and produces one output protein that all have the same molecule signal. By connecting two *NOT* gates to a wired *OR* gate in this manner, the wired *OR* gate acts as a wire that ties the output protein from the two *NOT* gates.

Wired *OR* gates are generated in this workflow after genetic gates are enriched by VPR and have been identified into their proper gate types. All generated *NOT* gates are iterated through to build wired *OR* gates with input and output signals set to the same molecule signal. These wired *OR* gates can then export into SBOL library files that can be used for the technology mapping procedure.

If these wired *OR* gates are provided to **Algorithm 4.1** for identifying its gate type, then they are returned as a wired *OR* gate and not a standard *OR* gate. This type of gate is recognized in the algorithm as a wired *OR* gate because the gate's input and output all have the same molecule signal. Categorizing these wired gates in this manner also helps resolve crosstalk when these types of gates are used in the technology mapping procedure mentioned in Chapter 5.

If a given *gateMD* does not follow these expected gate structures that are presented in **Figure 4.4**, then the gate is categorized as a *NOTSUPPORTED* gate. The *NOTSUPPORTED* gates can be utilized in the technology mapping procedure by providing a structural Verilog of the gate so that the gate can be interpreted in the procedure. This finalizes the gate identifying procedure. A user building their library of gates can then decide whether these sorted gates can be exported in separate SBOL files or into a single SBOL file that they can then use for the technology mapping procedure mentioned in Chapter 5.

4.4 Summary

This chapter demonstrates a programmatic methodology for generating genetic gates. Large quantities of transcriptional units are assembled in SBOLDESIGNER using the concept of *CombinatorialDerivation*. The generated transcription units are enriched with genetic interaction information using VPR. The enriched transcription units are then processed and identified as different gate types. This automated procedure was used to build two genetic gate libraries. These generated libraries are used for testing the work presented in Chapter 7.

CHAPTER 5

TECHNOLOGY MAPPING FOR ASYNCHRONOUS GENETIC CIRCUITS

The workflow described in this dissertation uses technology mapping, as shown in **Figure 1.2**, to realize a physical design from a gate-level design (netlist). Technology mapping, as applied to genetic circuits, is a class of optimization problem that aims to get an optimized design following a predefined scoring system while avoiding genetic constraints discussed in **Figure 4.1**. In the context of this dissertation, the goal is to minimize the sequence of the genetic circuit design. The technology mapping procedure presented in this workflow is adapted from Roehner et al. [103]. This dissertation incorporates all of the technology mapping algorithms in [103], including branch and bound, greedy, and exhaustive algorithms.

The technology mapping presented in this dissertation is different from the work presented in Roehner et al. This procedure builds around an asynchronous circuit design workflow. The technology mapping procedure takes a synthesized design using the procedures described in Section 3.2 and selects technology-specific gates generated in Section 4.2 while avoiding genetic constraints. The proposed technology mapping supports more logic gates. Namely, the proposed workflow allows the use of arbitrary gates by using *NOTSUPPORTED* gates. *NOTSUPPORTED* gates referred to in Section 4.3 are not identified as one of the core gates supported in this workflow. A user has the option to provide a Verilog file that defines the gate's behavior in order to incorporate *NOTSUPPORTED* gates. The proposed technology mapping also supports memory gates, which are required for constructing sequential circuits.

This chapter is organized as follows. Section 5.1 describes a decomposition method applied to the synthesized design before technology mapping and the reasons why this step is necessary in achieving better results. Then, the technology mapping procedure used in the workflow proposed in this dissertation is described. The technology mapping procedure is composed of two key steps: matching and covering. Section 5.2 goes over the implementation of the matching algorithm and Section 5.3 goes over the implementation for the covering algorithm and some variants. Details on handling genetic constraints are discussed throughout the matching and covering steps. Section 5.4 shows the process to generate a netlist of gates into SBOL. Lastly, Section 5.5 concludes the chapter with a summary of the highlights.

5.1 Boolean Decomposition

The ATACS tool produces a canonical normal form for the synthesized design using an SOP composed of *NOT*, *OR*, and *AND* gates. However, using technology mapping on the SOP representation of the logic circuit limits the solution set. The solution set can be expanded by using decomposition, where the structural design produced by the ATACS tool and the genetic logic gates from the generated gate library are decomposed into equivalent logic using only *NOT* and *NOR* gates. Decomposition is useful to find better gate mapping that can possibly yield better design results.

Using *NOT* and *NOR* gates is advantageous because a *NOR* gate is a universal gate that can be used to build any Boolean logic circuit. As shown on **Figure 5.1**, *OR* gate is represented by a series of *NOR* gates connected to a *NOT* gate. A 2-input *AND* gate involves attaching two 1-input *NOT* gates as inputs to a 2-input *NOR* gate. A *NAND* gate has a 2-input *NOR* gate with its input and output signals inverted with 1-input *NOT* gates. Section 3.2.4 has a detailed explanation on how individual gates can be decomposed into *NOR* and *NOT* gates. In addition to the gates described in Section 3.2.4, *NOTSUPPORTED* gates are also decomposed. Gates identified as *NOTSUPPORTED* are decomposed by performing synthesis on the gate and then applying logic minimization using the YOSYS tool on the resulting structural design. Logic minimization is performed due to a limited number of gates in the gate library and genetic constraints limiting the number of gates that can be used together. The decomposed gate is then parsed and converted into the SBOL data format using the Verilog parser implemented for the work described in Section 3.2.4.

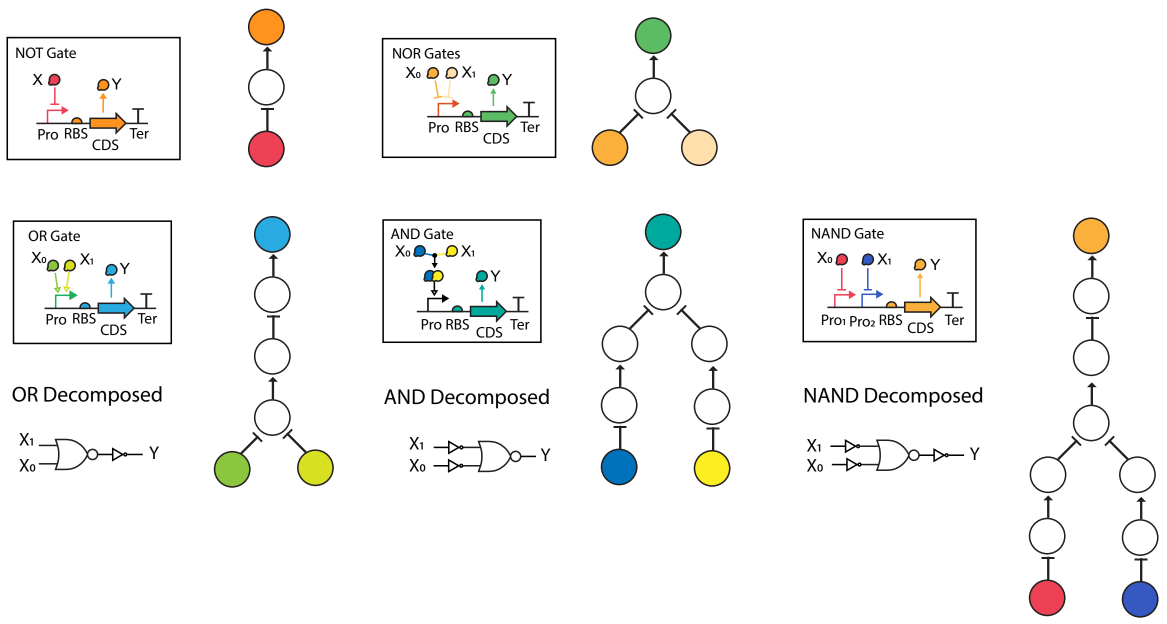


Figure 5.1: A representation of the logic gates decomposed to a graph-based format before performing technology mapping. A 2-input *AND* gate involves attaching two 1-input *NOT* gates as inputs to a 2-input *NOR* gate. A *NAND* gate has a 2-input *NOR* gate with its input and output signals inverted with 1-input *NOT* gates. *NOT* and *NOR* gates, on the other hand, are left in their original form. The graph-based structure shown on the right of each genetic gate represents a *DecomposedGraph*. Circles represent nodes. A root node has no parent. A leaf node has no child. Interactions are used to form connections between nodes. Arrows with pointy heads indicate production and arrows with flat heads are repression. Color represents signal carrier type. A white *DecomposedGraphNode* indicates that there is no signal carrier type assigned. A *DecomposedGraphNode* can have a cost factor that is calculated by DNA sequences.

While logic gates and circuits used in the workflow are represented in SBOL, the technology mapping procedure uses a custom representation for the decomposed nodes. A graph-based structure is used to encode the decomposition of these genetic gates and specification. The graph-based structure, shown in **Figure 5.1**, is referred to in the implementation as a *DecomposedGraph*. A *DecomposedGraph* has a list of *DecomposedGraphNodes* that represents the components used for assembling a genetic gate. A *DecomposedGraphNode* marked as a leaf node represents an input of a gate, and a *DecomposedGraphNode* marked as a root node represents an output of a gate. A *DecomposedGraphNode* can be assigned with a signal carrier type based on the color of the nodes shown in **Figure 5.1**. *DecomposedGraphNodes* with the same color indicate that the signal carrier types are the same. A white *DecomposedGraphNode* indicates that there is no signal carrier type assigned. A

DecomposedGraphNode connects to another *DecomposedGraphNode* by adding a *NodeInteractionType*. *NodeInteractionType* uses repression and production as ontology terms to define the relationship between connecting *DecomposedGraphNodes*.

Figure 5.2 shows the mapping of SBOL objects to the objects described above. That is, *ModuleDefinition* objects are converted into *DecomposedGraph* objects, *FunctionalComponent* objects are converted into *DecomposedGraphNode* nodes, and *Interaction* objects are converted into NodeInteractionType objects. A *DecomposedGraphNode* is set to a leaf if a *FunctionalComponent* has its *Direction* set to *DirectionType.IN* and a *DecomposedGraphNode* is set to an output node if a *FunctionalComponent* has its *Direction* set to *DirectionType.OUT*. The *Type* of each *Interaction* within a *ModuleDefinition*, represented as an SBO [142] term, is converted into *NodeInteractionType*. If the *Type* contains *SystemsBiologyOntology.INHIBITION*, then the converted *NodeInteractionType* is set to *NodeInteractionType.REPRESSION*. The direction of the node interactions is determined by locating the *DecomposedGraphNodes* for the *Participants* involved in the *Participation* of an *Interaction*. If the *Participant* has the *Role INHIBITED*, then the *DecomposedGraphNode* for this *Participant* is set as the parent for its *NodeInteractionType*. The child of this *NodeInteractionType* is set to the *DecomposedGraphNode* with the *Participant* containing the *Role* of *INHIBITOR*. The same concept applies when an *Interaction* with *Type SystemsBiologyOntology.GENETIC_PRODUCTION* is encountered. If an *Interaction* has *Type SystemsBiologyOntology.GENETIC_PRODUCTION*, the node relationship is set to *NodeInteractionType.PRODUCTION*. The *DecomposedGraphNode* representing the parent of this *NodeInteractionType* is the *Participant* containing the *Role* of *PRODUCT*. The *DecomposedGraphNode* representing the child of this *NodeInteractionType* is the *Partici-*

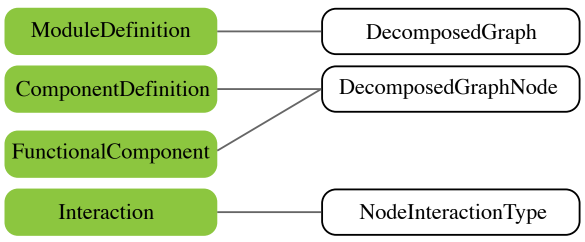


Figure 5.2: A figure to illustrate how an SBOL data model is converted into a *DecomposedGraph*. This conversion transforms the given structural specification to a graph-based structure before performing technology mapping so that genetic gates can map to the specification. *NOTSUPPORTED* gates can also use this conversion if a behavioral Verilog was provided and synthesized into a structural representation.

pant containing the *Role* of *TEMPLATE*.

A *DecomposedGraphNode* has a score. On a library of genetic gates, the score set on the root *DecomposedGraphNode* acts as the cost (in DNA sequence length) for using the gate. On a specification, the score set on each *DecomposedGraphNode* serves as a scoreboard to keep track of the best designs. Effectively, the score of each node indicates the DNA sequence length of the sub-circuit up to the node. A *DecomposedGraphNode* also has a feature to allow a user to preselect a specific *ComponentDefinition* when performing technology mapping. This feature provides a user the capability to indicate what specific molecule must pair to a particular *DecomposedGraphNode*, which can improve runtime since it constrains the solution space. A caveat to this preselected feature is the URI obtained from the given *ComponentDefinition* must exist as a primary input or a primary output within one of the libraries of genetic gates provided for technology mapping. If the library of genetic gates does not contain *ComponentDefinitions* with this same URI, the technology mapping procedure will not find a solution. The use of these features is explained in further detail in Section 5.3.

Once the library of genetic gates and the structural design of the specification are decomposed, the technology mapping procedure can then proceed to the matching step. For the remainder of this chapter, the matching and covering steps are discussed using *DecomposedGraph* as the internal data structure for performing technology mapping.

5.2 Matching

Matching entails filtering through a list of library gates and recording which genetic gate is capable of covering a subset of the *DecomposedGraph* for the design specification. While the matching step in Roehner et al. [103] requires the specification to be a directed acyclic graph (DAG)(i.e., only combinational circuits), the matching step described in this chapter supports feedback loops, and thus sequential circuits. The algorithm is described in **Algorithm 5.1**. The algorithm takes a *DecomposedGraph*, *S*, representing the design specification and a gate library, *G*, as inputs. The algorithm outputs *M*, which maps each node in *S* to a list of gates that can be used to cover that particular node.

The first step in matching is to indicate that all nodes in *S* have not been traversed by assigning the nodes with a score of ∞. Topological sort is then called on *S* to get the order

Algorithm 5.1: Match

```
1  Input: Specification $S$, Gate Library $G$;
2  Output: Map of node matches $M$ ;
3  setAllGraphNodeScore($S, \infty$);
4  $N$ = topologicalSort($S$);
5  foreach $n$ in $N$ do
6      foreach $g$ in $G$ do
7          if isMatch($n$, $g$.root) then
8              insert($M, n, g$);
9              totalScore = score($g$) + score($n$) ;
10             if $totalScore < score(n)$ then
11                 setScore($n$, totalScore);
12     if $\neg contains(M, n)$ then
13         setScore($n$, 0);
14 return $M$;
```

that *DecomposedGraphNodes* should be visited when pairing genetic gates. The topological sort implemented in this matching step is ordered from leaf nodes to the root node. The sorted nodes are stored in list N. Each node n in N checks if a genetic gate, g, provided in the library G can be used to cover that particular subset of the specification using **Algorithm 5.2**. If there is a match, then gate g is inserted into the output M as a match for node n. The score of node n is also computed when selecting the gate. This is used as a bound in the covering method. A gate, g, can pair to a *DecomposedGraphNode*, n, in S if all *DecomposeGraphNodes* of g and the *InteractionTypes* connecting these *DecomposedGraphNodes* match the structure of n. Since the order of the inputs matters when mapping gates to the specification, this matching process checks for different corner cases (described later on) when pairing genetic gates to the specification.

As shown in **Algorithm 5.2**, a set *visited* is created to catalog the *DecomposedGraphNode* encountered when checking if the structure of a genetic gate matches the structure of the specification's *DecomposedGraphNode*. This set also eliminates infinite loops when traversing *DecomposedGraphNodes* with feedback. The queue indicates the order in which each *DecomposedGraphNode* on the specification is visited and matched. During initialization, the queue adds the root *DecomposedGraphNode* of s and g. Initializing the queue to start at the root *DecomposedGraphNode* of s and g indicates where the matching pattern begins. *DecomposedGraphNode* for s and g are continually added to this queue until the leaf nodes

Algorithm 5.2: isMatch

```
Inputs: Specification node $s$, Genetic Gate $g$;
Output: Boolean;
visited = $\{\}$;
queue = $\langle\rangle$;
insert($queue, s, g$);
while $\neg queue.empty$ do
    curr_s, curr_g = queue.pop;
    if $isPreselected(curr\_s)$ then
        if $\neg isNodeCdMatch(curr\_s, curr\_g)$ then
            return false;
    if $size(curr\_s.children) \neq size(curr\_g.children)$ then
        return false;
    if $size(curr\_s.children)) = 1 \wedge interactionMatch(curr\_s, curr\_g)$ then
        if $isPreselected(curr\_s.child1) \wedge hasCdAssigned(curr\_g.child1)$ then
            if $isNodeCdMatch(curr\_s.child1, curr\_g.child1)$ then
                if $\neg contains(visited, curr\_s.child1)$ then
                    add($visited, curr\_s.child1$);
                    add($queue, curr\_s.child1, curr\_g.child1$);
        else if $\neg contains(visited, curr\_s.child1)$ then
            add($visited, curr\_s.child1$);
            add($queue, curr\_s.child1, curr\_g.child1$);
    else if $size(curr\_s.children) = 2 \wedge interactionMatch(curr\_s, curr\_g)$ then
        if $isPreselected(curr\_s.children)$ then
            match2inputPreselect($queue, visited, curr\_s.children, curr\_g.children$);
        else
            if $\neg contains(visited, curr\_s.child1)$ then
                add($visited, curr\_s.child1$);
                add($queue, curr\_s.child1, curr\_g.child1$);
            if $\neg contains(visited, curr\_s.child2)$ then
                add($visited, curr\_s.child2$);
                add($queue, curr\_s.child2, curr\_g.child2$);
    if $\neg visited.containsAll(curr\_s.children())$ then
        return false;
return true;
```

for *g* are encountered. Matching a node *g* to a node *s* begins by removing the first pairing of nodes *s* and *g* from the queue and storing each node to *curr_s* and *curr_g*, respectively. Then, requirements that define if *g* matches *s* are checked in the following order. First, the *ComponentDefinition* URI for *curr_s* and *curr_g* is checked for equivalency if the *curr_s* has been assigned with a *ComponentDefinition* URI. Second, the number of children for *curr_s* and *curr_g* is checked to see if they are the same size. Third, properties of the children node for *curr_s* and *curr_g* are checked before adding the children nodes to the queue to continue matching the next set of nodes on *s* and *g*. In the case that one child connects to *curr_s* and *curr_g* and the interactions that connect the parent to the child node on *s* and *g* match, then two corner cases are checked. The first corner case examines if the first child of *curr_g* has been preselected with a molecule signal. If the signal assignment on *curr_s.child*1 matches with the signal on *curr_g.child*1, then queue adds *curr_s.child*1 and *curr_g.child*1 and visited adds *curr_s.child*1 if visted has not encountered *curr_s.child*1. The second corner case applies when *curr_s.child*1 does not have a *ComponentDefinition* preselected. In this scenario, the *curr_s.child*1 adds to the queue if it has not been visited.

Similar corner cases apply when there are two children connected to *curr_s* and *curr_g* and their interactions connected on *curr_s* are the same as they are on *curr_g*. If the children nodes for *curr_s* are preselected, then the signals assigned to the children nodes are checked for equivalency with the children nodes of *curr_g*. These signal checks are performed in **match2inputPreselect**. In this scenario, there are four corner cases that are checked before the children nodes for *curr_s* and *curr_g* are added to the queue.

The first corner case applies when *curr_s.child*1 is preselected and *curr_s.child*2 has not been preselected. An input signal for *curr_g* in this case must match the signal assigned to *curr_child*1. If there is an input signal on *curr_g* that matches the signal assigned to *curr_s.child*1, then visited and queue are updated with the children nodes for *curr_s* and *curr_g* if visited does not contain *curr_s.child* and *curr_s.child*2. The second corner case is an inverse of the first corner case and follows the same checks. The third corner case applies when *curr_s.child*1 and *curr_s.child*2 are preselected with a molecule signal. In this scenario, both input signals for *curr_g* must match the signals assigned to the two children of *curr_s*. This scenario involves checking the assignment of *curr_g.input*1 and *curr_g.input*2 in two different orientations. If one orientation does not match, the pairing of signals is reversed to ensure that the matching of *g* on *s* is not dependent on the layout of *g* paired to *s*. The fourth corner case occurs when *curr_s.child*1 and *curr_s.child*2 are not preselected with molecule signals. In this case, the queue and visited updates if *curr_s.child*1 and *curr_s.child*2 have not been visited. False returns from **isMatch** when the children for *curr_s* were not all encountered. True returns from **isMatch** when queue is empty and all nodes in *g* have been traversed and checked with *s*.

If **Algorithm 5.2** returns true for a node pair, g and n, then the pair is inserted into M. The score of n is also updated if the total score of g summed with n is less than the current score of n. The score of g is determined based on summing the length of all *Sequences* referenced by all *ComponentDefinitions* that compose the genetic gate. Note that DNA sequence length is used as the score because sequence length of a DNA has shown correlations to the delay of transcription and translation [103]. Nonetheless, there are many other ways to score a design. Chapter 8 proposes other relevant genetic circuit parameters that could be incorporated for future work. In the case that no gates can map to n, then the score of n is set to zero. This matching step continues until all *DecomposedGraphNodes* in N have been visited. The solution returned from this match algorithm is then used in the covering step.

In order to better understand how the matching procedure works, an example is used. **Figure 5.3** shows an example of a technology mapping problem, where the specification is a decomposed AND gate and five genetic gates are provided in the gate library. The goal is to select gates that realize the specification while minimizing cost. The cost of each gate is calculated by the sum of sequences stored on the *DecomposedGraphNodes*. The color

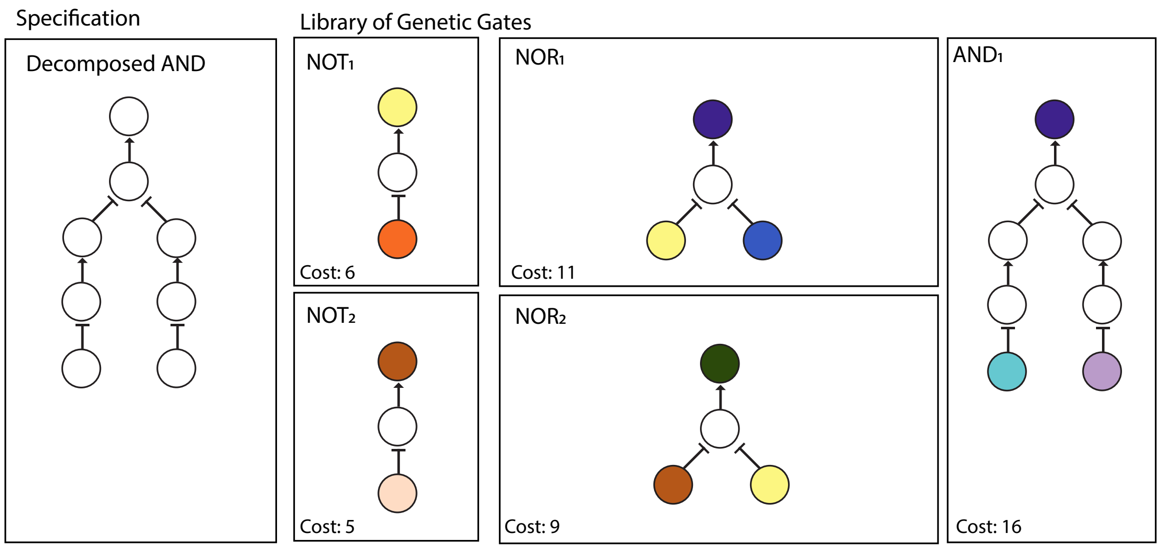


Figure 5.3: An example of the technology mapping procedure taking in two forms of inputs. The first is a specification describing an AND gate. The second is a library containing five genetic gates. The cost of each gate is calculated by the sum of sequences stored on the *DecomposedGraphNodes*. The color code assigned to each genetic gate's *DecomposedGraphNode* indicates the assignment of a molecule signal. If *DecomposedGraphNodes* have the same color assigned, then this indicates that molecule signals are the same.

code assigned to each genetic gate's *DecomposedGraphNode* indicates the assignment of a molecule signal. If *DecomposedGraphNodes* have the same color assigned, then this indicates that molecule signals are the same.

Figure 5.4 shows the result of the technology mapping procedure after calling the matching step on the AND specification. In this step, the library of genetic gates is mapped onto the AND specification. Genetic gates that can map to the specification are recorded on the specification's *DecomposedGraphNodes*. Where a genetic gate maps on the specification's *DecomposedGraphNode* corresponds to the gate's root *DecomposedGraphNode*.

5.3 Covering

The map of node M produced from the matching procedure is then used to perform the covering procedure. Covering obtains an optimal solution that realizes the specification. There are three covering methods that are supported in the proposed workflow: exhaustive, greedy, and branch and bound. The number of optimal solutions varies depending on the selected algorithm.

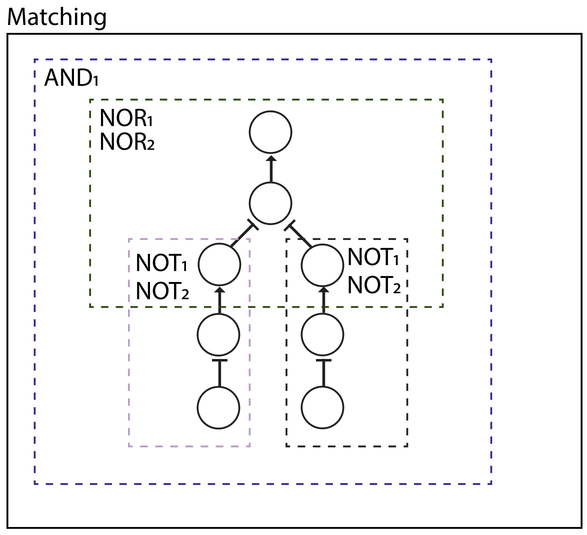


Figure 5.4: The result of the matching step when applied on the AND specification. In this step, the library of genetic gates are mapped onto the AND specification. Genetic gates that can map to the specification are recorded on the specification's *DecomposedGraphNodes*. Where a genetic gate maps on the specification's *DecomposedGraphNode* corresponds to the gate's root *DecomposedGraphNode*.

5.3.1 Exhaustive Covering Algorithm

The exhaustive covering algorithm is a brute force methodology for finding and enumerating every possible solution. Exhaustive is performed by calling **Algorithm 5.3** to generate all possible solutions. **Algorithm 5.3** takes a map of node matches M, a number of solution γ, and a Boolean flag $sortMatches$ to produce a list of solution α. When calling exhaustive, γ is set to MAX_VALUE to specify that the list of solutions returned from **Algorithm 5.3** should contain all possible solutions generated for the desired specification. $sortMatches$ is set to false to indicate that genetic gates selected at each node in specification should not depend on the order in which gates are sorted when building a solution.

Algorithm 5.3 begins with $queueOfSol$ and α as two empty lists. $queueOfSol$ stores the order of possible solutions that are evaluated and α stores complete solutions that are found when performing $baseCover$. S represents the specification that the matching procedure used for pairing genetic gates and it is used in this covering algorithm for generating solutions from those genetic gates that pair to the specification. β is a technology mapping solution initialized to zero. The root node of S adds to β as an unmappedNode to indicate the beginning node on S that a genetic gate can map onto. Then, β adds to $queueOfSol$ as the first solution that needs to be evaluated. As long as $queueOfSol$ is not empty, then the first technology mapping solution is evaluated by removing from $queueOfSol$ and storing to β. If β has unmappedNodes that are needed to assign genetic gates to, then **nextUnmappedNode** retrieves the β and stores it into s. Genetic gates that map to s are retrieved from getGateList using M to look up the node s and the list of gates are then stored into $gateList$. Each gate g is evaluated at node s to ensure constraints that determine if a g can include in solution β by checking for the following conditions. The first condition is to check if s has been assigned with a *ComponentDefinition* by calling **hasCdAssigned** and if the output signal of g matches the assigned *ComponentDefinition*value. This type of check ensures molecule signals must be the same in order to link the input and output signals of g to its neighboring gates assigned in β. **Figure 5.5** shows NOT_1 and NOT_2 as two possible genetic gates that can map to the specification node encased within the dashed rectangular box. The color assigned to each node in the *DecomposedGraphNodes* indicates the molecule signal assigned to that node. Nodes that have the same color indicate that the same molecule signal is assigned. In this example, NOT_1 can assign to the yellow node in

Algorithm 5.3: baseCover

```
Inputs: Map of node matches $M$, Number of Solutions $\gamma$, Boolean sortMatches;
Output: List of Solutions $\lambda$;
queueOfSol = $\langle\rangle$;
$\alpha$ = $\langle\rangle$;
S = $M.specification$;
$\beta = \emptyset$;
setScore($\beta, 0$);
addUnmappedNode($\beta$, S.root);
add(queueOfSol, $\beta$);
while $\neg queueOfSol.empty$ do
    $\beta$ = queueOfSol.pop;
    while hasUnmappedNode($\beta$) do
        s = nextUnmappedNode($\beta$);
        gateList = getGateList($M, s$);
        if $gateList.empty$ then
            continue;
        if sortMatches then
            sortAscendingOrder($M, gateList$);
        foreach $g$ in $gateList$ do
            if $hasCdAssigned(s) \wedge getAssignedComponent(s) \neq g.output$ then
                continue;
            $\lambda = \beta$;
            if isRoot($S, s$) then
                assignNode($s, g.output$);
            next_s = getMatchingEndNodes($s, g$);
            if $size(next\_s) = 1$ then
                Cover_1input($\lambda, s, next\_s, g, queueOfSol$);
            else if $size(next\_s) = 2$ then
                Cover_2input($\lambda, s, next\_s, g, queueOfSol$);
            if $isSolutionComplete(S, \beta)$ then
                if $\neg hasCrosstalk(\beta)$ then
                    if $size(\alpha) < \gamma$ then
                        insert($\alpha, \beta$);
                    else
                        return $\alpha$;
    return $\alpha$;
```

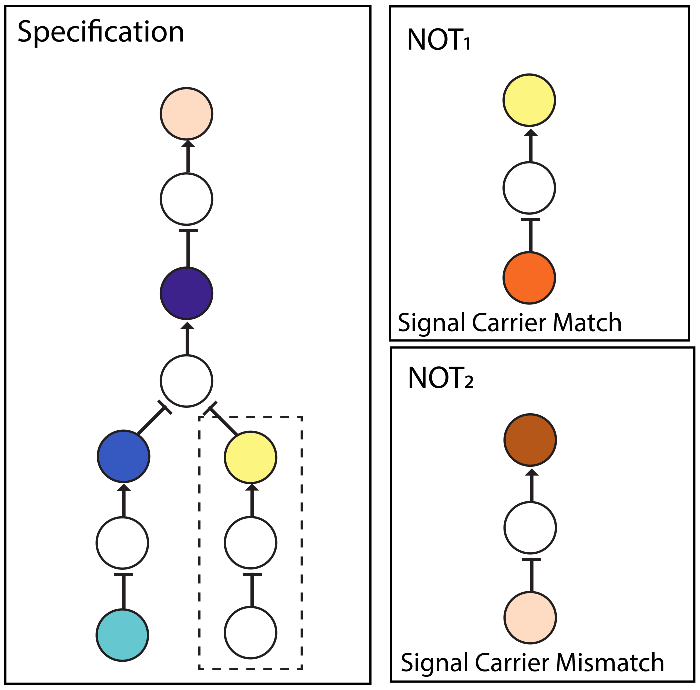


Figure 5.5: An example to show how signal carrier mismatch is determind when covering a genetic gate to the specification. A genetic gate is added to the current solution when the molecule signal on a root *DecomposedGraphNode* of a genetic gate matches the molecule signal assigned to a specification's *DecomposedGraphNode*. In this example, the brown *DecomposedGraphNode* for NOT_2 does not match the yellow *DecomposedGraphNode* assigned on the specification. However, NOT_1 has the same molecule signal that matches the assigned specification's *DecomposedGraphNode* and is thus added to the current solution for covering.

the specification because NOT_1 has the same molecule signal assigned to its output node. NOT_2, however, cannot assign to the yellow node on the specification because its output node does not match in signal type.

If the signal matches from the output value of g to s, then the solution, β, is copied over and stored into λ to ensure that each solution evaluated at node s for different gate g is not overwritten. The second condition is to check if s is a root node of S, then the output signal of g is assigned to s. This check ensures that the root node s is not left empty without a molecule assignment. The third check is to look at the children node where g ends on S and observe if input signals for g match signals at S. The children nodes where the input signals of g map on S are computed by calling **getEndNodes**. **getEndNodes** takes in two *DecomposedGraphNodes* to indicate where the root node of g maps to the node of s and returns a list of *DecomposedGraphNodes* where g ends on s. **Figure 5.6** shows

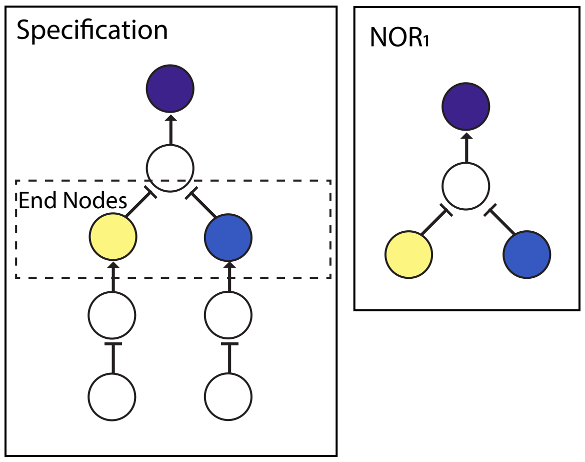


Figure 5.6: The purpose of EndNodes is used for matching the *DecomposedGraphNodes* for g paired to s. EndNodes are calculated by determining where the root of a genetic gate begins and where the genetic gate's leaf *DecomposedGraphNodes* end on the specification. In this figure, the yellow and blue *DecomposedGraphNodes* indicate where NOR_1 ends on the given specification. The EndNodes calculated in this example is where the yellow and blue *DecomposedGraphNodes* map on the specification *DecomposedGraphNodes*.

an example of **getEndNodes** performed on NOR_1 mapped to the specification. In this example, the purple nodes on NOR_1 and the specification are provided to **getEndNodes** as the starting point where NOR_1 maps to the specification. The yellow and blue nodes on the specification are used to indicate where the leaf nodes of NOR_1 end on the specification and are returned from this method as the children nodes of s. These children nodes are stored in *next_s*.

If the size of *next_s* is one, then **Algorithm 5.4** is called to check if the one input gate matches the signal assigned at *next_s* before adding g to λ. $totalScore$ represents the total score of λ if g is added to λ as a part of the solution. Then, the first node in *next_s* is evaluted to see if it has been assigned with a *ComponentDefinition* at λ. If there is no *ComponentDefinition* assigned to $next\_s.first$, then the gate is added to the λ and assigned to node s. The score for λ is also updated with the $totalScore$ to reflect the addition of a new gate added to the solution. Then, $next\_s.first$ is assigned with the input signal of g with a *ComponentDefinition*. λ is also added back onto $queueOfSol$ to continue on generating a complete solution for the specification. In the case that *next_s.first* has been assigned with a *ComponentDefinition*, then the input signal of g is compared to $next\_s.first$. If the *ComponentDefiniton* matches, then the g maps to s and adds to λ. The score of λ is also

Algorithm 5.4: Cover_1input

```
1  Input: Map of Node $\lambda$, Specification Node $s$, List of Specification Nodes $next\_s$,
   Genetic Gate $g$, List of Solution queueOfSol;
2  totalScore = score($\lambda$) + score($g$);
3  if $\neg isNodeMapped(\lambda, next\_s.first)$ then
4      addGate($\lambda, s, g$);
5      setScore($\lambda, totalScore$);
6      assignNode($\lambda, next\_s.first, g.input1$);
7      addUnmappedNode($\lambda, next\_s.first$);
8      add($queueOfSol, \lambda$);
9  else
10     if $isNodeCdMatch(next\_s.first, g.input1)$ then
11         addGate($\lambda, s, g$);
12         setScore($\lambda, totalScore$);
```

updated with $totalScore$ to reflect the additon of a g added to λ.

If the size of the children nodes for $next\_s$ is two, then **Algorithm 5.5** is called. Because there are two nodes that the input signals for g can assign to $next\_s$, then each node for $next\_s$ must check if it can pair with both input signals of g. There are four scenarios that could happen during this condition. The first scencario is when $next\_s.second$ is not assigned with a *ComponentDefinition* and $next\_s.second$ is assigned with a *ComponentDefinition*. In this case, one input signal of g must match a *ComponentDefinition* signal assigned to $next\_s.second$. If the signal matches, then **Algorithm 5.6** is called so that g can map to s and is added to the solution λ. The score of λ is updated to reflect the addition of g to λ. The next unmapped node added to λ is $next\_s.first$. Then, the new solution for λ is added onto $queueOfSol$ to continue covering the procedure.

The second scenario occurs when $next\_s.first$ is assigned and $next\_s.second$ is not assigned with a *ComponentDefinition*. In this case, the same condition applies from the first scenario but $next\_s.second$ is added to λ as the unmapped node. The third scenario occurs when $next\_s.first$ and $next\_s.second$ are assigned with a *ComponentDefinition*. In this case, both input signals of g must match the *ComponentDefinition* assigned to the two nodes in $next\_s$. The last scenario occurs when both nodes in $next\_s$ are not assigned with a *ComponentDefinition*. In this case, $sol1$ and $sol2$ are two technology mapping solutions created to test different orientation of the input signals for g assigned to $next\_s$. The solution for $sol1$ represents the assignement of $next\_s.first$ assigned to $g.input1$ and $next\_s.second$ assigned

Algorithm 5.5: Cover_2input

```
Input: Map of Node $\lambda$, Specification Node $s$, List of Specification Nodes $next\_s$, Genetic Gate $g$, List of Solution queueOfSol;
if $\neg isNodeMapped(\lambda, next\_s.first \wedge isNodeMapped(\lambda, next\_s.second))$ then
    if $isNodeCdMatch(\lambda, next\_s.second, g.input1)$ then
        updateCovSol($queueOfSol, \lambda, s, g, totalScore, next\_s.first, g.input2$);
    else if $isNodeCdMatch(\lambda, next\_s.second, g.input2)$ then
        updateCovSol($queueOfSol, \lambda, s, g, totalScore, next\_s.first, g.input1$);
else if $isNodeMapped(\lambda, next\_s.first) \wedge \neg isNodeMapped(\lambda, next\_s.second)$ then
    if $isNodeCdMatch(\lambda, next\_s.first, g.input1)$ then
        updateCovSol($queueOfSol, \lambda, s, g, totalScore, next\_s.second, g.input2$);
    else if $isNodeCdMatch(\lambda, next\_s.second, g.input2)$ then
        updateCovSol($queueOfSol, \lambda, s, g, totalScore, next\_s.second, g.input1$);
else if $isNodeMapped(\lambda, next\_s.first) \wedge isNodeMapped(\lambda, next\_s.second)$ then
    if $isNodeCdMatch(\lambda, next\_s.first, g.input1) \wedge$
     $isNodeCdMatch(\lambda, next\_s.second, g.input2)$ then
        addGate($\lambda, s, g$);
        setScore($\lambda, totalScore$);
    else if $isNodeCdMatch(\lambda, next\_s.first, g.input2) \wedge$
     $isNodeCdMatch(\lambda, next\_s.second, g.input1)$ then
        addGate($\lambda, s, g$);
        setScore($\lambda, totalScore$);
else
    sol1 = $\lambda$;
    addGate($sol1, s, g$);
    setScore($sol1, totalScore$);
    assignNode($sol1, next\_s.first, g.input1$);
    assignNode($sol1, next\_s.second, g.input2$);
    addUnmappedNode($sol1, next\_s.first$);
    addUnmappedNode($sol1, next\_s.second$);
    add($sol1, queueOfSol$);
    sol2 = $\lambda$;
    addGate($sol2, s, g$);
    setScore($sol2, totalScore$);
    assignNode($sol2, next\_s.first, g.input1$);
    assignNode($sol2, next\_s.second, g.input2$);
    addUnmappedNode($sol2, next\_s.first$);
    addUnmappedNode($sol2, next\_s.second$);
    add($sol2, queueOfSol$);
```

Algorithm 5.6: updateCovSol

```
1 Inputs: List of Solution queueOfSol, Map of Node $\lambda$, Specification Node $s$, Genetic
  Gate $g$, Score of Solution totalScore, Specification Nodes $next\_s$, Gate signal $gs$;
2 addGate($\lambda, s, g$);
3 setScore($\lambda, totalScore$);
4 assignNode($\lambda, next\_s, gs$);
5 addUnmappedNode($\lambda, next\_s$);
6 add($queueOfSol, \lambda$);
```

to $g.input2$. Gate g is then assigned to s and added to $sol1$. The score for $sol1$ is updated and both $next\_s.first$ and $next\_s.second$ are added as the next set of nodes that are used to perform covering on $sol1$. $sol1$ is also added to the $queueOfSol$ as a possible solution. $sol2$ is similar to $sol1$ except the input signal for g assigned to $next\_s.first$ and $next\_s.second$ are inverted.

After checking the signals assigned to $next\_s$, the last check is to see if the current solution for β is complete by calling **isSolutionComplete**. **isSolutionComplete** takes the specification S and the solution β to iterate over all genetic gates assigned to their corresponding s node. If the genetic gates completely cover S, then true is returned. Otherwise, false is returned. Once a complete solution has been found, then β is examined for crosstalk. Recall that crosstalk occurs when there is interference of circuit components with each other or the host circuitry. **Algorithm 5.7** shows the implementation to check for crosstalk. **Algorithm 5.7** begins by creating an empty *signals* that is used for storing *ComponentDefinitions* that are assigned in S. Then, all nodes s are iterated in S. The *ComponentDefinition* assigned to s in the solution β is retrieved by calling **getAssignedComponents** and storing the value in cd. If a cd is assigned to s, then the signal for cd is checked to see if it is

Algorithm 5.7: hasCrosstalk

```
1 Input: Specification S, Map of Node $\beta$ ;
2 Output: Boolean;
3 signals = {};
4 foreach $s$ in $S$ do
5     cd = getAssignedComponent(s, signals);
6     if $\neg empty(cd)$ then
7         if $contains(cd, signals)$ then
8             return true;
9 return false;
```

contained within *signals*. True is returned, in this case, if *signals* already contains *cd*. False is returned when each *cd* stored in *signals* is unique.

An illustration to show how crosstalk is considered in the covering step is shown in **Figure 5.7**. In this example, NOT_2, if added to the current solution, causes crosstalk with an existing genetic gate that has already been selected in the solution. The *DecomposedGraphNodes* causing crosstalk in this example is the tan colored *DecomposedGraphNode* located in the specification conflicting with the tan colored *DecomposedGraphNode* in NOT_2. NOT_1, if added to the current solution, avoids crosstalk because the colors assigned to each *DecomposedGraphNode* in the specification are unique.

If **Algorithm 5.3** identified that the solution β does not have crosstalk, then β is added to α if the size of α is less than the number of solution specified as γ. Because exhausting

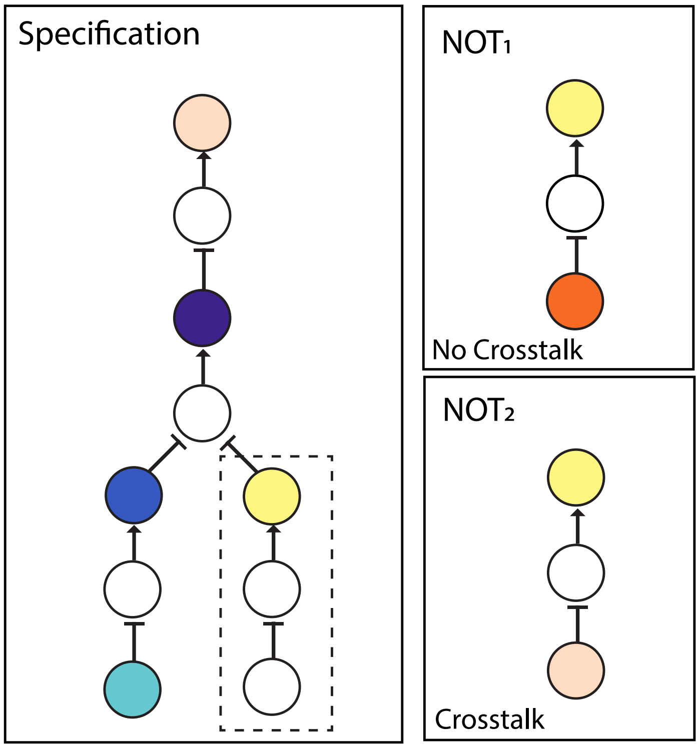


Figure 5.7: An example to show how crosstalk is computed in the covering step. In this example, NOT_2, if added to the current solution, causes crosstalk with an existing genetic gate that has already been selected in the solution. The *DecomposedGraphNodes* causing crosstalk in this example is the tan colored *DecomposedGraphNode* located in the specification conflicting with the tan colored *DecomposedGraphNode* in NOT_2. NOT_1, if added to the current solution, avoids crosstalk because the colors assigned to each *DecomposedGraphNode* in the specification are unique.

generates all solutions that can map to S, each solution β will continue to add to α until all possible solutions have been explored. This covering routine stops when all solutions are evaluated in $queuOfSol$.

One of the problems of the exhaustive method is its complexity. Because all possible solutions are enumerated, the time it takes to find all solutions results in $O(n!)$, where n represents the number of *DecomposedGraphNodes* in the specification. Factorial of n results from enumerating multiple solutions at each specification's *DecomposedGraphNode* with a different selection of genetic circuit.

5.3.2 Greedy Covering Algorithm

Greedy is a covering algorithm designed to find a solution by choosing genetic gates with the lowest score at each step performed in the covering method. Genetic gates selected in this covering algorithm are locally optimal but not globally optimal. The drawback of this algorithm is that the selection of one genetic gate affects the decision of future genetic gates. Although a genetic gate selected at the beginning of the covering algorithm may have low cost, this restricts future gate coverings. These restricted genetic gates could produce a covering solution with a better score than the one computed by the greedy algorithm. However, this covering method is advantageous to use when the specification has optimal substructures and the library of genetic gates is limited to the choices that can pair to the specification.

Presented in this work is a greedy algorithm designed to produce the first n number of solutions found, where n represents a finite integer value. The greedy covering algorithm is similar to the exhaustive algorithm because it calls the same **baseCovering**. The difference in implementation is specifying γ to be an n value provided by the user running this technology mapping procedure. The last difference is setting the $sortMatches$ to true so that the genetic gate selected at each node s is always evaluating g with the lowest score first.

5.3.3 Branch and Bound Covering

Branch and bound is a covering algorithm that uses scores from genetic gates as a cost factor to find the best solution with the lowest cost for generating a netlist of genetic gates. Branch and bound is advantageous to use when looking for an optimal solution that best satisfies all the constraints that are considered in the method. **Algorithm 5.8** is an implementation of the branch and bound algorithm. This algorithm takes a specification S to perform the covering method on, a specification node s to map a genetic gate, a map of node β to store the current technology mapping solution, and a map of node M that stores the list of genetic gates mapped at each node in S. This algorithm produces a technology mapping solution α that represents a netlist with the lowest score summed by all genetic gates selected to form a complete cover for S.

This algorithm begins by retrieving all genetic gates mapped at s from M and storing the list of genetic gates in *gateList*. In the case that there is no gate mapped at s, then this indicates that s is at the end of the specification S. When this happens, the solution for β is checked to see if the solution found at that instant is complete, has crosstalk, and if the score of β is better than the score of α. If these three conditions are satisified, then β is returned as the best solution. Otherwise, the existing best solution for α is returned.

Algorithm 5.8: BranchBoundRecurse

```
1  Inputs: Specification $S$, Specification Node $s$, Map of Node $\beta$, Genetic Gate $prev\_g$
   Map of node matches $M$;
2  Output: Map of Node $\alpha$ ;
3  gateList = getGates($M, s$);
4  if $\neg gateList.empty$ then
5      if $isSolutionComplete(S, \beta) \wedge \neg hasCrosstalk(S, \beta) \wedge score(\beta) < score(\alpha)$ then
6          return $\beta$;
7      else
8          return $\alpha$;
9  foreach $g$ in $gateList$ do
10     estimatedScore = score($g$) + score($\beta$) + score($s$);
11     if $estimatedScore >= score(\alpha)$ then
12         continue;
13     $\lambda$ = $\beta$;
14     addGate($\lambda, s, g$);
15     setScore($\lambda, score(\lambda) + score(g)$);
16     if $isRoot(S, s)$ then
17         assignNode($\lambda, s, g.output$);
18     next_s = getMatchingEndNodes($s, g$);
19     if $size(next\_s) = 1$ then
20         bbCover_1input($\alpha, \lambda, S, s, next\_s, gM$);
21     else if $size(next\_s) = 2$ then
22         bbCover_2input($\alpha, \lambda, S, s, next\_s, g, M$);
23 return $\alpha$;
```

If there are genetic gates assigned to s, then each gate g is evaluated to find the optimal solution for α. *estimatedScore* stores a predicted value for β if the remaining gates selected from s down to the leaf nodes of S are gates selected in the solution with the lowest score. If *estimatedScore* is less than the best score α, then g is evaluated in the cover. The solution for β is stored in λ so that each gate g assigned to s for a solution β is not overwritten when backtracking to evaluate for alternative solutions. After storing the solution for β in λ, g is added to λ and the score of λ is incremented with the score of g. If s is a root node of S, then the output signal of g is assigned to s in λ. This check is important when checking for crosstalk in order to ensure that all signals involved in connecting genetic gates to cover S are not ignored. Then, the end nodes where g ends on S are evaluated and stored in $next\_s$. If the number of nodes stored in $next\_s$ is a size of one, then this indicates that the gate assigned to s is a 1-input gate. Likewise, if the number of nodes stored in $next\_s$ is a size of two, then this indicates tht the gate assigned to s is a 2-input gate.

If the gate g assigned to s is a 1-input gate, then the input of g is checked to see if it matches the signal assigned to $next\_s$ by calling **Algorithm 5.9**. **Algorithm 5.9** checks to see if there is a signal assigned to the node at $next\_s.first$. If there is no signal assigned to $next\_s.first$, then the input signal for g is assigned to $next\_s.first$. The algorithm continues to perform a complete cover for λ by recursively calling itself with $next\_s.first$ as the next starting node evaluated. Otherwise, if the input signal of g does not match the signal assigned to $next\_s.first$, then the solution for λ is ignored by setting its score to ∞. If g assigned to s is a 2-input gate, then the 2-input signals for g are checked to see if they match with the signals assigned to $next\_s$ by calling **Algorithm 5.10**. If one of two nodes in $next\_s$ are assigned with a signal, then the first two condition shown in **Algorithm 5.10**

Algorithm 5.9: bbCover_1input

```
1 Inputs: Map of node $\alpha$, Map of Node $\lambda$  Specification $S$, Specification node $s$, List of
  Specification node $next\_s$, Genetic Gate $g$, Map of node matches $M$;
2 if $\neg isNodeMapped(\lambda, next\_s.first)$ then
3     assignNode($\lambda, next\_s.first, g.input1$);
4     $\alpha$ = BranchBoundRecurse($S, next\_s.first, \lambda, g, M$);
5 else
6     if $\neg isNodeCdMatch(\beta, next\_s.first, g.input1)$ then
7         setScore($\lambda, \infty$);
```

Algorithm 5.10: bbCover_2input

```
Inputs: Map of Node $\alpha$, Map of Node $\lambda$, Specification Node $s$, List of Specification Node $next\_s$, Genetic Gate $g$, Map of Node matches $M$;
if $\neg isNodeMapped(\lambda, next\_s.first) \wedge isNodeMapped(\lambda, next\_s.second)$ then
    if isNodeCdMatch($next\_s.second, g.input1$) then
        assignNode($\lambda, next\_s.first, g.input2$);
        $\alpha$ = BranchBoundRecurse($S, next\_s.first, \lambda, g, M$);
    else if isNodeCdMatch($next\_s.second, g.input2$) then
        assignNode($\lambda, next\_s.first, g.input1$);
        $\alpha$ = BranchBoundRecurse($S, next\_s.first, \lambda, g, M$);
    else
        setScore($\lambda, \infty$);
else if $isNodeMapped(\lambda, next\_s.first) \wedge \neg isNodeMapped(\lambda, next\_s.second)$ then
    if isNodeCdMatch($next\_s.first, g.input1$) then
        assignNode($\beta, next\_s.second, g.input2$);
        $\alpha$ = BranchBoundRecurse($S, next\_s.second, \lambda, g, M$);
    else if isNodeCdMatch($next\_s.first, g.input2$) then
        assignNode($\lambda, next\_s.second, g.input1$);
        $\alpha$ = BranchBoundRecurse($S, next\_s.second, \lambda, g, M$);
    else
        setScore($\lambda, \infty$);
else if $isNodeMapped(\lambda, next\_s.first) \wedge isNodeMapped(\lambda, next\_s.second)$ then
    if $\neg isNodeCdMatch(next\_s.first, g.input1) \wedge$
      $\neg isNodeCdMatch(next\_s.second, g.input2)$ then
        setScore($\lambda, \infty$);
    else if $\neg isNodeCdMatch(next\_s.first, g.input2) \wedge$
      $\neg isNodeCdMatch(next\_s.second, g.input1)$ then
        setScore($\lambda, \infty$);
else
    sol1 = $\lambda$;
    assignNode($sol1, next\_s.first, g.input1$);
    assignNode($sol1, next\_s.second, g.input2$);
    $\alpha$ = BranchBoundRecurse($S, next\_s.first, sol1, g, M$);
    $\alpha$ = BranchBoundRecurse($S, next\_s.second, sol1, g, M$);
    sol2 = $\lambda$;
    assignNode($sol2, next\_s.first, g.input2$);
    assignNode($sol2, next\_s.second, g.input1$);
    $\alpha$ = BranchBoundRecurse($S, next\_s.first, sol2, g, M$);
    $\alpha$ = BranchBoundRecurse($S, next\_s.second, sol2, g, M$);
```

are executed to match one of 2-input signals of g. If a matching signal has been found for an input signal of g paired to a node in *next_s*, then the node in *next_s* for the solution λ is assigned with the matching input signal. Then, covering is performed on the other node in *next_s* if the node has not been covered yet. When both nodes in *next_s* are assigned with a signal, then a check is made to see if both input signals for g match the nodes for *next_s*. The score of λ is set to ∞ if all possible assignments that g can pair to *next_s* do not match. The last condition occurs when the nodes in *next_s* have not been assigned with a signal. In this scenario, λ is assigned to $sol1$ and $sol2$ to evaluate different possible solutions for which the input signals for g can pair to the nodes of *next_s*. Each solution is evaluated by assigning the corresponding input signals g to the nodes in *next_s*. Then, a recursive call is performed for both nodes in *next_s* to ensure that all nodes linked to the nodes in *next_s* are covered with genetic gates. The solutions returned from each recursive call are stored back onto α to ensure that the best solution is always up to date with the best solution.

Figure 5.8 shows two possible solutions generated after the covering step for the AND specification. If exhaustive was selected, then both solutions from this figure are returned from the technology mapping procedure. If greedy was selected, then the solution returned from this procedure varies based on the number of solutions provided as the input to this covering algorithm. If the number of solutions provided to the method is one, then the solution with a cost of 20 is returned. The solution with a cost of 20 is returned from this method because greedy is designed to select gates with the lowest score for covering a specification's *DecomposedGraphNode*. In this particular example, there are three genetic gates that could map to the specification's root *DecomposedGraphNodes*. Of the three possible genetic gates, NOR_2 with the cost of 9 was selected because it has the lowest cost. The remaining genetic gates needed for completing the specification are NOT_1 and NOT_2. As a result, the solution with a cost of 20 is returned. If branch and bound was selected, then the final solution returned from this method is the solution with a cost of 16. This solution was selected from branch and bound because out of both solutions that were found, this solution has the lowest score. **Figure 5.9** shows the technology mapping procedure applied to an example with a feedback loop. In this example, SR Latch is described as the specification and the four genetic gates represent the library. The result of the matching step has paired NOT_1 and NOT_2 to the OUT and NOR_1 and NOR_2 to Q

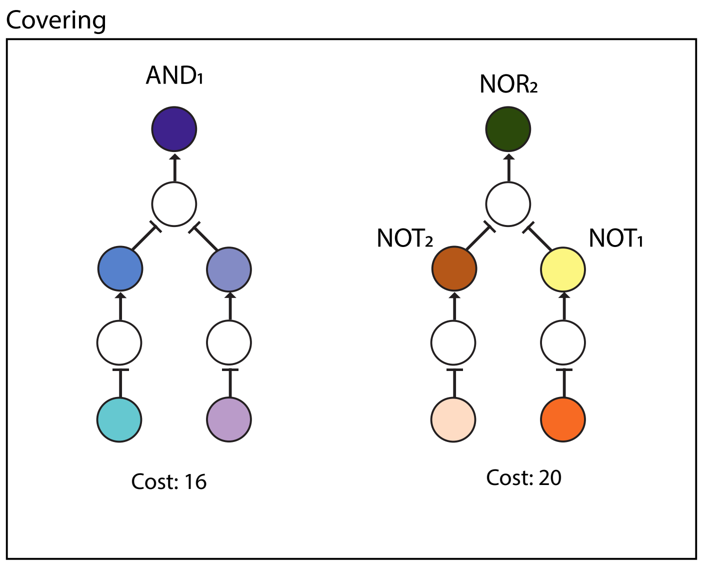


Figure 5.8: A figure showing the result of the technology mapping procedure after calling the covering step on the AND specification. Two solutions are derived in this figure after performing the covering step. Signal carrier mismatch and crosstalk are accounted for when generating these solutions. If exhaustive was selected for the running of the covering step, then both solutions from this figure are returned from the technology mapping procedure. If greedy was selected, then the solution returned from this procedure varies based on the number of solutions provided as the input to this covering algorithm. If the number of solutions provided to the method is one, then the solution with a cost of 20 is returned. Greedy selects this solution with a cost of 20 to return from its method because greedy is designed to select gates with the lowest score for covering a specification's *DecomposedGraphNode*. If branch and bound was selected, then the final solution returned from this method is the solution with a cost of 16. This solution was selected from branch and bound because out of both solutions that were found, this solution has the lowest score.

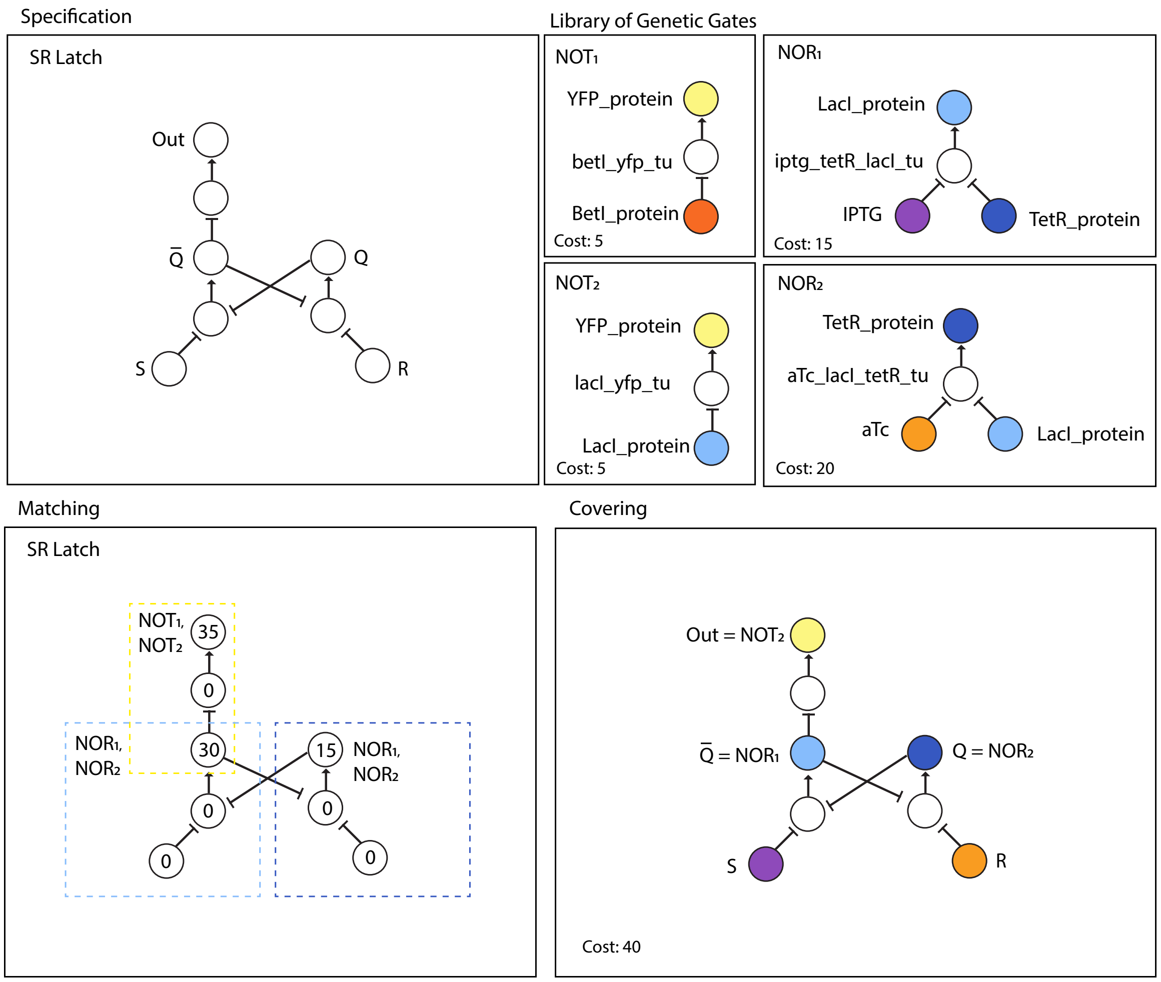


Figure 5.9: An example of the technology mapping procedure performed on the SR Latch example. The four genetic gates provided as the library are generated from the CELLO SYNBIOHUB collection. The matching step identified NOT_1 and NOT_2 mapped to Out and NOR_1 and NOR_2 mapped to $\bar{Q}$ and Q. One solution is produced from the covering step because all three covering algorithms identified that NOT_1 results in signal carrier mismatch if it is added to the solution for technology mapping.

and $\bar{Q}$. Each genetic gate selected in the covering method is guaranteed to connect when the input and output signals share the same signals with their connecting genetic gates. As a result, the covering step generated one solution because all three covering algorithms identified that NOT_1 results in signal carrier mismatch if it is added to the solution for technology mapping.

5.4 Generating a Netlist of Genetic Gates Encoded into SBOL

The technology mapping procedure presented in this chapter is incorporated in this workflow for designing asynchronous genetic circuits by converting the netlist from technology mapping into SBOL. **Algorithm 5.11** shows the implementation to generate SBOL from a given covering solution along with the specification S that the solution was derived from. The final SBOL format returned from this algorithm is an *SBOLDocument* represented as δ. This algorithms begins by initializing δ to an empty document. A map of connections to molecules, *moleculeConnections*, is created to keep track of the molecules connected from a genetic gate to its neighboring gate. Then a *ModuleDefinition*, *circuit*, is created to connect gates assigned to nodes s in δ. **Algorithm 5.12** collects input and output signals and their connection to other gates in α. **Algorithm 5.12** takes the specification, S, and the covering solution, α, to produce ϱ as a map of genetic gates assigned to a *FunctionalComponent* for linking genetic gate input and output signals. σ is a

Algorithm 5.11: generateSbolNetlist

```
1  Inputs: Specification $S$, Cover Solution $\alpha$;
2  Output: SBOLDocument $\delta$;
3  $\delta = \emptyset$;
4  moleculeConnection = $\langle\rangle$;
5  circuit = createMd($\delta, S$);
6  $\varrho$ = getTopLevelConnections($S, \alpha$);
7  foreach $gc$ in $\varrho$ do
8      g_instance = addGate(circuit, gc.gate);
9      foreach $c$ in $gc.connections$ do
10         if $\neg contains(moleculeConnection, c)$ then
11             molecule = createFc(circuit, c.cd);
12             insert(moleculeConnection, c, molecule);
13         circuit_molecule = getMolecule($moleculeConnection, c.cd$);
14         createConnection($g\_instance, circuit\_molecule, c.cd$);
15 return $\delta$;
```

Algorithm 5.12: getTopLevelConnections

```
Inputs: Specification $S$, Cover Solution $\alpha$;
Output: Gate Connections $\varrho$;
$\varrho = \langle\rangle$;
$\sigma = \langle\rangle$;
visited = {};
queue = $\langle\rangle$;
s = $S$.root;
add(queue, s);
while $\neg queue.empty$ do
    s = queue.pop;
    g = getGate($\alpha, s$);
    if $g \neq \emptyset$ then
        gate_c = createGateConnection($g$);
        foreach $i$ in $g.inputs$ do
            if contains($\sigma$, $i$) then
                c = getMoleculeConnection($\sigma, i$);
                if isDirectionOut(c) then
                    setDirection($c, DirectionType.INOUT$);
                addGateConnection($gate\_c, c$);
                addConnection($c, g, i$);
            else
                c = createMoleculeConnection($i$);
                setDirection($c, DirectionType.IN$);
                addGateConnection($c, g, i$);
        foreach $o$ in $g.outputs$ do
            if contains($\sigma$, $o$) then
                c = getMoleculeConnection($\sigma, o$);
                if isDirectionIn(c) then
                    setDirection($c, DirectionType.INOUT$);
                addGateConnection($gate\_c, c$);
                addConnection($c, g, o$);
            else
                c = createMoleculeConnection($o$);
                setDirection($c, DirectionType.OUT$);
                addConnection($c, g, o$);
        next_s = getMatchingEndNodes($s, g.root$);
        foreach $n$ in $next\_s$ do
            insert(queue, n);
        add($\varrho, gate\_c$);
return $\varrho$;
```

list of connections that molecule signals are involved in. *visited* is a list created to prevent an infinite loop from occurring when traversing nodes in S. *queue* is also a list that stores the order in which nodes in S are evaluated. ϱ, σ, and *visited* are initially empty while the root node of S is added to *queue* as the starting point to record molecules and their related connections.

Each node in *queue* is evaluated and stored in s. The variable g represents the gate assigned to s from the solution α. A genetic gate g is empty when there is no gate assigned to s. If a genetic gate g exists for s, then the input and output signals of g are retrieved to store the signals and their connection to their corresponding gate to *gate_c*. A connection, c, stores information about a signal and the gate that contains that signal. A connection c is created for each input and output signal of g. The variable ϱ checks if a connection c has been created for each input signal i and for each output signal o. If ϱ contains the given signal, then the connection for the given signal is retrieved and stored in c. If this signal is an input and it was marked as a primary output or an output marked as a primary input in a previous connection, then genetic gates sharing this signal are set to an $INOUT$ signal to indicate that this signal is not a primary input or primary output signal of *circuit*. The gate g and the signal are added onto the new connection c. c is then added into *gate_c* to record the signal connections that exist for each g. If a signal does not exist in ϱ, then a new connection c is created. Then, the signal and the gate is added to c. Once connections are recorded for all input and output signals of a gate g, then the next nodes, *next_s*, where g ends on S are evaluated by adding the nodes onto *queue*. *gate_c* adds to σ to store gates that were evaluated for signal connections. This procedure continues until all nodes in s are traversed.

Algorithm 5.11 takes the connections generated from **Algorithm 5.12** and stores them into ϱ. Each genetic gate included in the solution α is added to *circuit* by going over each connection, *gc*, from ϱ. The signal, *c.cd* assigned to a connection c from *gc* is created as a *FunctionalComponent* in *circuit* as *molecule*, if the signal does not exist within *moleculeConnection*. If the signal, *c.cd*, already exists in *circuit* as a *FunctionalComponent*, then it is retrieved from *circuit* and stored to *circuit_molecule*. Then, a *MapsTo* object is created to connect *circuit_molecule* to the signal connection *c.cd*. This process continues untill all connections are explored for every gate assigned to S.

Figure 5.10 shows an SR Latch example converted into SBOL from a netlist of genetic gates that were generated from **Figure 5.9**. The *topLevelCircuit* is the SR Latch. The genetic gates that were selected in the covering solution are instantiated in the SR Latch as *Modules*. These *Modules* are shown in the figure as NOT_1, NOR_1, and NOR_2. IPTG, aTc, TetR_protein, LacI_protein, and YFP_protein are *FunctionalComponents* that are instantiated on the *topLevelCircuit* as the input and output signals of the genetic gates. The connections that are formed from a gate's input and output signals to the instantiated signals on the SR Latch are done so through *MapsTo* objects.

5.5 Summary

Technology mapping is a procedure that takes in a specification and a library of genetic gates to realize a circuit. This procedure converts a structural design into a netlist of genetic gates described in the form of SBOL. The technology mapping discussed in this chapter is performed in two stages: matching and covering. This chapter describes these two steps and discusses three algorithms for covering, including exhaustive, greedy, and branch and bound. There are many ways to perform covering, especially when accounting for different

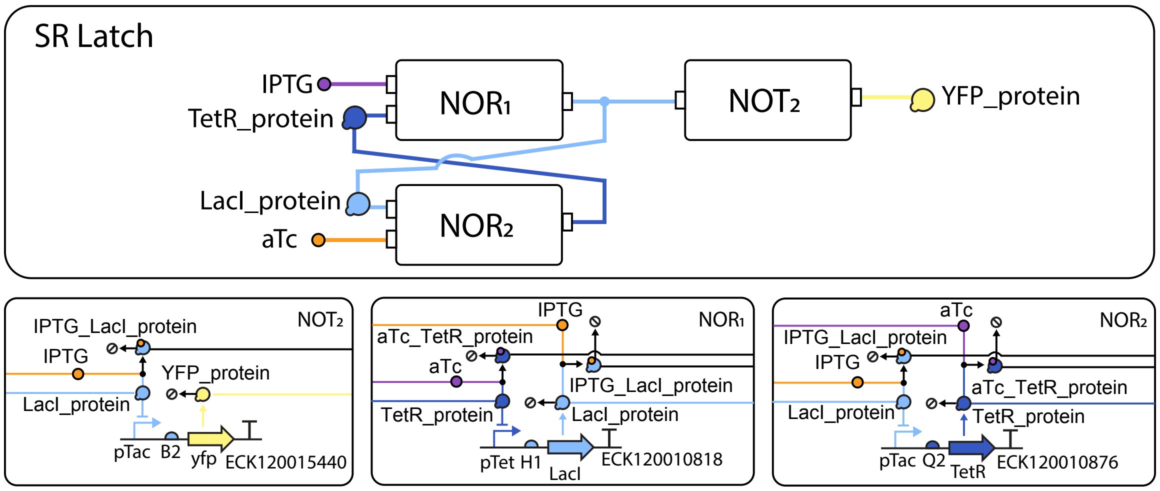


Figure 5.10: A netlist of the SR Latch represented in SBOL. The genetic gates that were selected in the covering solution are instantiated in the SR Latch as *Modules*. These *Modules* are shown in the figure as NOT_1, NOR_1, and NOR_2. IPTG, aTc, TetR_protein, LacI_protein, and YFP_protein are *FunctionalComponents* instantiated on the *topLevelCircuit* as the input and output signals of these genetic gates. The connections that are formed from a gate's input and output signals to the instantiated signals on the SR Latch are done so through *MapsTo* objects.

genetic constraints. Many tools have been developed for technology mapping purposes, as shown in **Table 5.1**. This table summarizes technology mapping features and compares different qualities of each tool, specifically, what form the specification is in, the type of library parts or gates that the tools provide, and if any data standards are supported to test the solutions produced from these tools.

This technology mapping procedure was initially adapted from Roehner et al. [103] to extend iBioSim2 to iBioSim3 to support the design of sequential genetic circuits. However, there were several changes that were made from Roehner et al. work that altered the work that was presented from Roehner et al. Specifically, this workflow supports Verilog to design a genetic circuit, whereas Roehner et al. uses GRN as the specification language. Roehner et al. supports crosstalk, whereas this workflow extends the support of genetic constraints by addressing roadblock when assembling transcriptional units to build genetic gates. Roehner et al. used a PoP-style method to form and connect genetic gates, whereas this workflow uses protein to protein interactions. The support of protein to protein interaction involves building genetic gates to use transcriptional units. Then, an enrichment procedure was used to retrieved molecules and their interactions from the SBH online repository to form the input and output signals of a gate. Gates identified in this gate generation process are supported to identify different structures that a genetic gate can take on to classify its logic behavior. Because protein to protein interaction are used in this workflow to connect signals between genetic gates, this style of connecting gates during the technology mapping procedure introduced signal carrier mismatch that must be addressed when connecting gates to form a netlist for the desired specification.

Table 5.1: This table summarizes the different existing technology mapping tools. It shows the tool's input specification language, standards supported, the types of library parts used, and the genetic constraints addressed.

	MATCHMAKER [66]	SBROME [64]	iBIOSIM2 [103]	CELLO [15]	GENETECH [104]	iBIOSIM3 [89]
Specification language	Abstract GRN	Abstract GRN	Abstract GRN	Verilog	Boolean expressions	Verilog
Library	Arbitrary logic gates	Parts	Arbitrary logic gates	*NOT* and *NOR* gates	*NOT* and *NOR* gates	Arbitrary logic gates
Signal Mismatch	✓	✗	✗	✓	✗	✗
Crosstalk	✗	✓	✓	✓	✓	✓
Context Effects	✗	✗	✗	✗	✗	✗
Roadblock	✗	✗	✗	✓	✗	✓
Standard Support	SBOL1	✗	SBOL2 and SBML	SBOL2	✗	SBOL2 and SBML
Supports Seq. Design	✗	✗	✗	✗	✗	✓

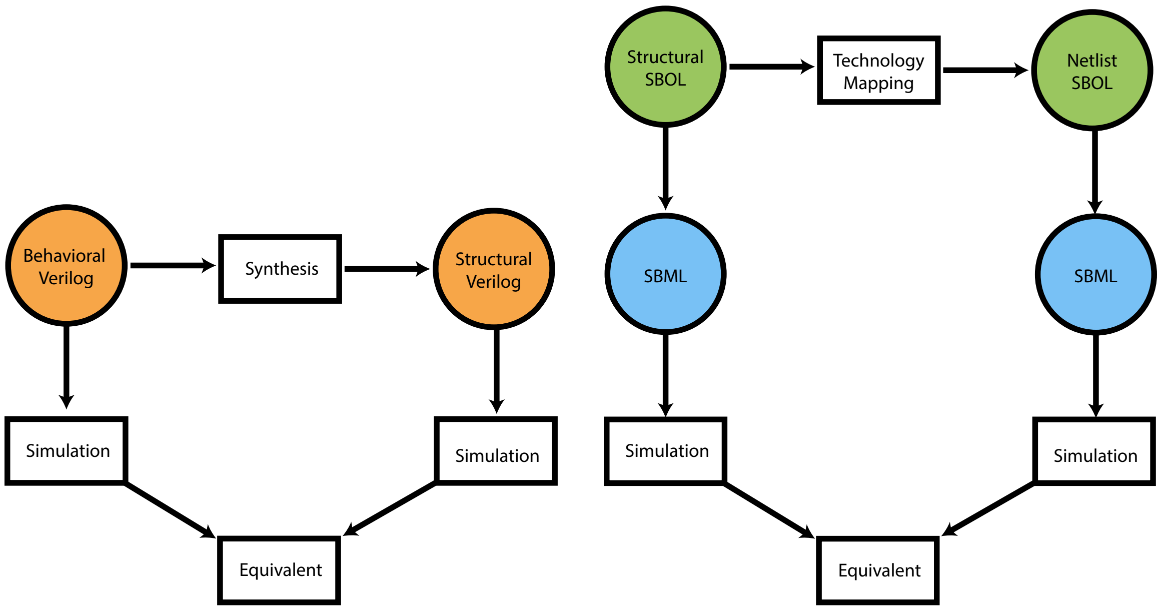
CHAPTER 6

VERIFICATION

The workflow described in this dissertation progresses through several procedures to translate a high-level description to a physical design composed of biological parts. The description of the design uses different formats to perform each task. Verification must be performed at each stage of the workflow to ensure that the behavior initially described in the design specification stays the same throughout the data format conversions.

Figure 6.1a and **Figure 6.1b** show the verification stages that are performed for this workflow. The outcome of this verification process is to ensure that the behavior that is observed in the simulation is the same in each step of the workflow. If the behavior observed in the simulation changes in one of these verification steps, then this is a clear indication that the intended behavior of the user's design has changed during the workflow. At this point, the Verilog compiler and/or converters must be fixed to ensure that information stays consistent and the data are producing valid results.

Simulation can perform when there is a modeling language used to describe a specification and its testbench. Verilog and SBML are the two modeling languages used in this workflow. Verilog simulations can be done by using any available Verilog simulation tool. For this workflow, ModelSim is selected to observe the behavioral and structural Verilog designs, whereas for genetic circuits, there are a variety of formal methods and computational tools for their analysis [150, 151]. iBioSim is selected as the simulation tool to model and simulate genetic circuits, since the tool is incorporated in this workflow and it supports data conversions to the SBML data format. From the array of simulation methods supported in iBioSim, stochastic simulation is chosen for analyzing the circuit's behavior. Stochastic simulation was chosen because the molecule counts in these circuits are low and the behavior of these genetic asynchronous circuits must be analyzed in the presence of noise and hazards.



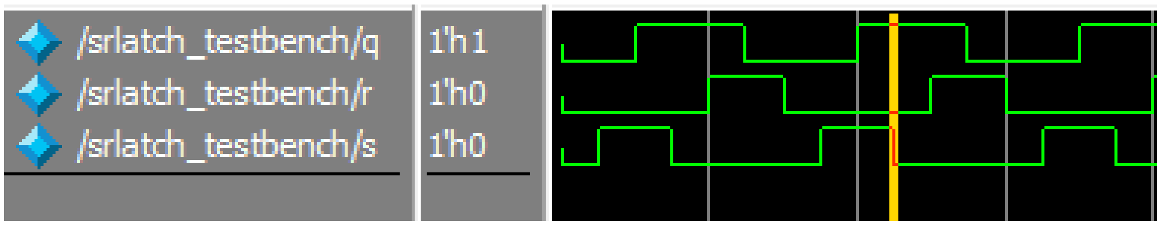
(a) Synthesis Verification (b) Genetic Circuit Verification

Figure 6.1: **Figure 6.1a** and **Figure 6.1b** are the two areas in the workflow where verifications are performed. Synthesis verification takes the behavioral Verilog and the structural Verilog produced from synthesis to perform simulation. The simulation results are verified to check if they are equivalent in simulation behavior. Genetic circuit verification is performed on the technology mapping procedure. This verification workflow takes the SBOL converted from the structural Verilog and the SBOL netlist produced from technology mapping. Both SBOL representations are compared and simulated within iBioSim by converting SBOL to SBML. The simulation results are compared and checked if they are equivalent in simulation behavior.

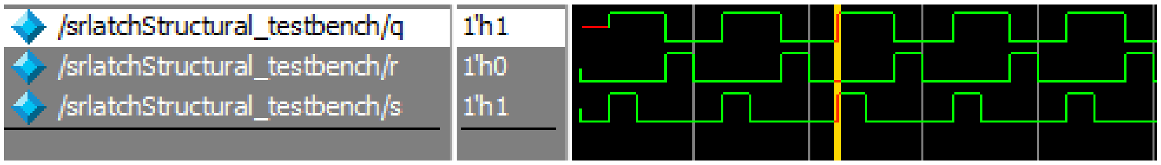
Work presented in Chapter 3 and Chapter 5 are the two areas in this workflow where verification is performed. The two sections in this chapter go into detail on the verification procedure for these two areas in the workflow. Section 6.1 describes the verification procedure performed when going from behavioral Verilog to structural Verilog. Section 6.2 describes verification performed on the genetic circuit generated from technology mapping.

6.1 Synthesis Verification

Chapter 3 demonstrated the process for translating a behavioral Verilog to a structural Verilog. **Figure 6.2** shows MODELSIM simulation results that are generated in this process. **Figure 6.2a** is a simulation generated using the specification in **Figure 3.2** and the testbench in **Figure 3.3** for testing the behavioral Verilog of an SR latch. **Figure 6.2b** shows a structural



(a)

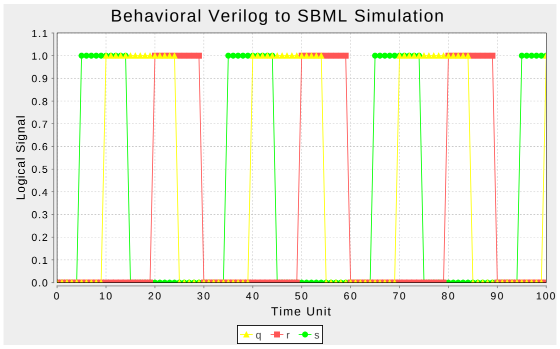

/srlatchStructural_testbench/q
1'h1
/srlatchStructural_testbench/r
1'h0
/srlatchStructural_testbench/s
1'h1

(b)

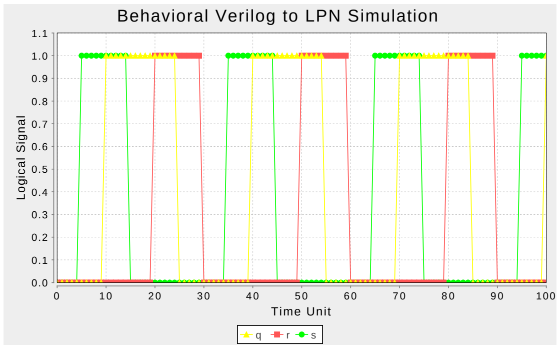
Figure 6.2: Simulation results for synthesis verification. **Figure 6.2a** shows a behavioral Verilog simulation of an SR latch executed in ModelSim. **Figure 6.2b** shows a structural Verilog simulation of an SR latch executed in ModelSim.

Verilog simulation of an SR latch. Note that the simulation results are not identifical but they are equivalent. More specifically, q goes HIGH when s goes HIGH and q goes LOW when r goes HIGH.

Figure 6.3 contains simulation results generated from iBioSim using stochastic simulation for testing the event-based SBML model and the LPN model presented in Section 3.2.1 and Section 3.2.2. Simulation on the event-based SBML model and the LPN model ensures that the behavior of the translated design is still behaving as expected before ATACS performs synthesis on the design. iBioSim supports LPN to SBML data conversion and thus



(a)



(b)

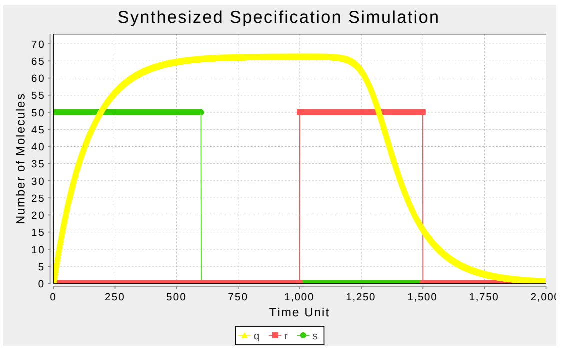
Figure 6.3: Simulation results for synthesis verification generated within iBioSim. **Figure 6.3a** shows a behavioral Verilog to SBML simulation of an SR latch. **Figure 6.3b** shows a behavioral Verilog to LPN simulation of an SR latch.

makes it possible to perform simulation if an LPN data model is provided to iBioSim. As shown in these simulations, the event-based SBML model and LPN model are simulating logic signals ranging from 0 and 1 as described. These signal values reflect the initial design specification where 0 bit and 1 bit data are randomly generated in the testbench.

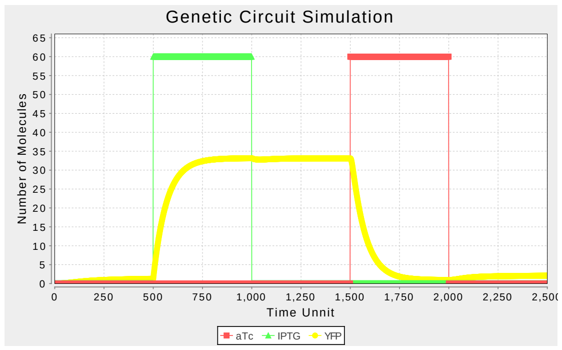
6.2 Genetic Circuit Verification

Chapter 5 is the last area where verification is performed. Simulations are generated within iBioSim by converting the SBOL files containing the structural design of the specification to SBML and the SBOL netlist converted into SBML after calling technology mapping.

As shown in **Figure 6.4**, the general idea of these two simulations is to ensure that the waveforms are all exhibiting the same behavior that is specified for the SR latch. Specifically, when r is LOW and s is HIGH, Q outputs a HIGH signal. Likewise, when r is HIGH and s is LOW, Q outputs a LOW signal. When r and s are LOW, then the previous output signal for Q is maintained. If the simulation shows a behavior that does not match the designed circuit, then this indicates the specific stage in the workflow that produces unexpected results. Then, this specific stage in the workflow is examined to resolve the issue. This issue could result from a corner case that was overlooked when designing the specification or a bug in the program. In either case, the simulation is used to track where this unexpected error occurs so that it can be resolved. Once the error is resolved, the next stage in the workflow is verified using the same simulation process until verification has



(a)



(b)

Figure 6.4: Simulation results for synthesis verification. **Figure 6.4a** shows a structural Verilog to SBOL simulation of an SR latch. **Figure 6.4b** shows a simulation of an SR latch after running technology mapping.

reached the end of the workflow.

Simulation is performed in iBioSim using stochastic simulation that indicates molecule amount dynamics rather than logic-based analysis since it better reflects the behavior of biological designs. Nonetheless, it is possible to see that the behavior is qualitatively equivalent to the logical design. One limitation when verifying the behavior of the produced model of biological designs from the SBOL to SBML conversion in iBioSim is that all parts use default kinetic parameter values. Because of these default values, the simulation result will also produce the same result. A future work in this area is to add a layer of characterization data on the SBOL designs produced from the technology mapping tool. These characterization data will be accessed by the VPR model generation procedure described by Misirili et al. [149]. This will, ideally, provide a more accurate depiction of the biological parts that were selected for the technology mapping procedure and how well it behaves in the circuit.

6.3 Summary

This chapter presents a verification procedure applied to stages of the workflow that involves going in and out of different data formats. Verification in this case refers to simulating Verilog and SBML for modeling and testing a given specification and its testbench. ModelSim and iBioSim are used as the two simulation tools. The goal of this verification process is to analyze the behavior of the simulations to see if the intended behavior of the designed circuit is the same across all simulations. Verification is complete at the end of the workflow when all simulations exhibit similar behaviors that are intended of the designed circuit.

CHAPTER 7

CASE STUDY: GENETIC LOW-PASS FILTER

This chapter presents a case study demonstrating this workflow on a genetic sensor that uses filtering and communication to make a more reliable detection decision. The library gates built for this example are presented in Section 7.1. Section 7.2 provides details on how the filter was created using the workflow presented in this dissertation. Section 7.3 goes over the results of our technology mapping procedure. Section 7.4 concludes with the advantages and limitations on the case study.

7.1 Cello Library Gates

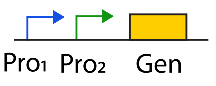
A gate library is generated from the CELLO *E. Coli* [15] collection from the SYNBIOHUB data repository. This collection was selected because it has information, such as DNA parts, molecules, and interactions that connect to these parts, that can be used to generate 1-input and 2-inputs gates. The CELLO dataset was built from the MIT/BU CELLO project to construct Boolean logic gates to perform technology mapping of combinational designs. The CELLO dataset has information about the different components to build Boolean logic gates and interactions on how the components relate to each other. Currently, CELLO has 72 DNA parts, 7 proteins, and 14 interactions stored on SYNBIOHUB. Although small, the CELLO dataset is valuable due to detailed information about its library of gates. CELLO's library of parts has a UCF file that specifies valid biological parts that can be constructed into NOT and NOR gates. This UCF file also has information on how genetic constraints are addressed when parts are selected to build the genetic combinational gates. Since the UCF file has been encoded into SBOL, CELLO's library of parts and the genetic constraint information are found on the SYNBIOHUB repository. Any genetic constraint that is specified to each biological part to assist during gate construction is attached to the components as annotations.

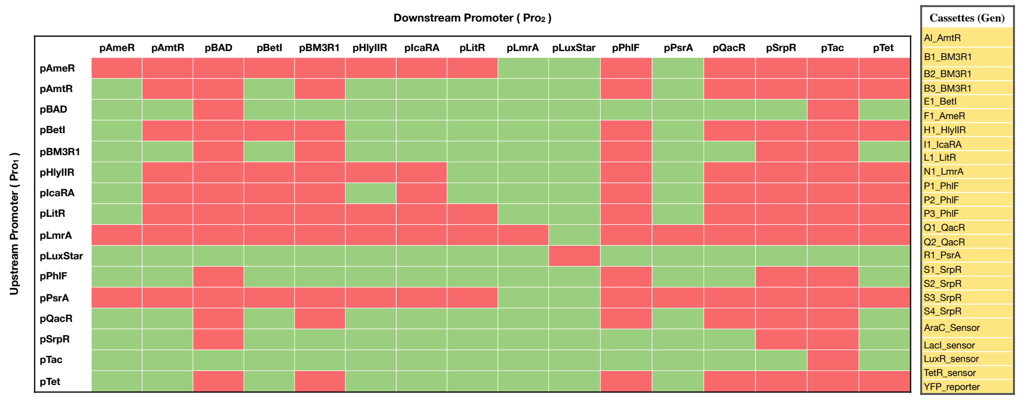
Two types of templates are created to generate transcriptional units. The first template has tandem promoters followed by an engineered region. The second template is a single promoter followed by an engineered region. The engineered region used in these two templates represents a cassette composed of DNA parts involved in the production of a targeted protein. These DNA parts include ribozymes, RBS, CDS, and terminator.

Figure 7.1 shows mapping of Cello DNA parts used for generating transcriptional units. In order to avoid roadblock, specific promoters are selected for the downstream promoters. The left table in the figure shows the different combinations that are used for a transcription unit with two promoters. The rows indicate promoters on position 1 and columns indicate promoters on position 2. Green indicates that the pairing is valid and red indicates the pairing is invalid. The right table in the figure shows the list of cassettes. After running VPR, there are 4,560 genetic gates that were generated from the CELLO collection. These gates were then identified and 2,696 are 2-input genetic $NAND$ gates, 40 are 2-input genetic AND gates, 274 are 1-input genetic NOT gates, 13 are 2-input genetic wired OR gates, and 1,537 are $NOTSUPPORTED$ gates.

7.2 Genetic Low-Pass Filter

This section presents a case study demonstrating our workflow on a genetic sensor that uses filtering and communication to make a more reliable detection decision. In particular,




Downstream Promoter (Pro₂)

Upstream Promoter (Pro₁)	pAmeR	pAmtR	pBAD	pBetI	pBM3R1	pHlyIIR	pIcaRA	pLitR	pLmrA	pLuxStar	pPhlF	pPsrA	pQacR	pSrpR	pTac	pTet
pAmeR	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟩	🟥	🟩	🟥	🟥	🟥	🟥
pAmtR	🟩	🟥	🟥	🟩	🟥	🟩	🟩	🟩	🟩	🟩	🟥	🟩	🟥	🟥	🟥	🟥
pBAD	🟩	🟩	🟥	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟩
pBetI	🟩	🟥	🟥	🟥	🟥	🟩	🟩	🟩	🟩	🟩	🟥	🟩	🟥	🟥	🟥	🟥
pBM3R1	🟩	🟩	🟥	🟩	🟥	🟩	🟩	🟩	🟩	🟩	🟥	🟩	🟩	🟥	🟥	🟩
pHlyIIR	🟩	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟩	🟩	🟥	🟩	🟥	🟥	🟥	🟥
pIcaRA	🟩	🟥	🟥	🟥	🟥	🟩	🟥	🟩	🟩	🟩	🟥	🟩	🟥	🟥	🟥	🟥
pLitR	🟩	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟩	🟥	🟩	🟥	🟥	🟥	🟥
pLmrA	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟥	🟥	🟥	🟥	🟥	🟥
pLuxStar	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟩	🟩	🟩	🟩	🟩	🟩
pPhlF	🟩	🟩	🟥	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟩	🟩	🟥	🟥	🟩
pPsrA	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟩	🟥	🟥	🟥	🟥	🟥	🟥
pQacR	🟩	🟩	🟥	🟩	🟥	🟩	🟩	🟩	🟩	🟩	🟥	🟩	🟥	🟥	🟥	🟩
pSrpR	🟩	🟩	🟥	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟥	🟩
pTac	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟩
pTet	🟩	🟩	🟥	🟩	🟥	🟩	🟩	🟩	🟩	🟩	🟥	🟩	🟥	🟥	🟥	🟥

Cassettes (Gen)
AI_AmtR
B1_BM3R1
B2_BM3R1
B3_BM3R1
E1_BetI
F1_AmeR
H1_HlyIIR
I1_IcaRA
L1_LitR
N1_LmrA
P1_PhlF
P2_PhlF
P3_PhlF
Q1_QacR
Q2_QacR
R1_PsrA
S1_SrpR
S2_SrpR
S3_SrpR
S4_SrpR
AraC_Sensor
LacI_sensor
LuxR_sensor
TetR_sensor
YFP_reporter

Figure 7.1: Mapping of Cello DNA parts used for building transcriptional units. The left table in the figure shows the different combinations that are used for a transcription unit with two promoters. The rows indicate promoters on position 1 and columns indicate promoters on position 2. Green indicates that the pairing is valid and red indicates the pairing is invalid. The right table in the figure shows the list of cassettes.

the sensor is constructed from three sensors as described in the Verilog specification and its testbench shown in **Figure 7.2** and **Figure 7.3**, respectively. These sensors are connected such that the Actuator of the first sensor is connected to the Start signal of the second sensor, and the Actuator of the second is connected to the Start signal of the third sensor. All three sensors share the same Sensor input. This genetic sensor circuit could be used, for example, to release pharmaceuticals in specific tissues, after sensing a cancer-related signal for a prolonged period of time [32, 152]. Examples of tissue-specific molecules, or the enzymes that produce them, are curated by the Human Protein Atlas and include surfactant protein A1 in the lungs, thyroglobulin in the thyroid, and uromodulin in the kidney [153–155]. There are many examples of possible cancer markers, though often elevation of a marker above normal is also used, examples include: HURP in prostate cancer cells [156], Carcinoembryonic antigen (CEA) in colorectal cancer [157], and CA 19-9 in pancreatic cancer [158]. The initial Start signal would be a tissue-specific chemical signature and the sensing input would be a cancer-related chemical. The output Actuator for the third sensor would be a chemotherapeutic agent. The bacteria would only produce the payload if they are in the correct tissue, and if they consistently sense the cancer-related input, thus filtering and decreasing the chance of false positives. Additionally, this pre-

```
module sensor_imp (Start, Sensor, Actuator);

        input wire Start, Sensor;
        output reg Actuator;

        initial begin
                Actuator = 1'b0;
        end

   always begin
      wait (Start == 1'b0 && Sensor == 1'b0);
      #5 Actuator = 1'b1;
      wait (Sensor == 1'b1);
      #5 Actuator = 1'b0;
   end

endmodule
```

Figure 7.2: Behavioral Verilog of a generalized C-element.

```
module sensor_testbench ();

        wire Actuator;
        reg Start, Sensor;


        initial begin
                Start = 1'b0;
                Sensor = 1'b0;
        end

        sensor_imp sensor_instance(
        .Start(Start),
        .Sensor(Sensor),
        .Actuator(Actuator)
        );

always begin
   #5 Sensor = 1'b0;
   #5 Start = 1'b0;
   wait (Actuator == 1'b1);
   #5 Sensor = 1'b1;
   wait (Actuator == 1'b0);
   #5 Sensor = 1'b0;
   wait (Actuator == 1'b1);
   #5 Start = 1'b1;
   #5 Sensor = 1'b1;
   wait (Actuator == 1'b0);
end

endmodule
```

Figure 7.3: The testbench of a generalized C-element.

vents the release of the chemotherapeutic agent in cancer-free tissues, reducing unwanted side effects.

To exemplify this work, a case study circuit is designed that uses IPTG as the Sensor input, aTc as the Start input, and *yellow fluorescent protein* (YFP) as the output. Each sensor is going to be transformed into a different cell type to reduce crosstalk concerns, allowing for gate reuse. Essentially, our circuit is composed of a population of three types of cells (note that the host cells may be identical but they each include a different circuit). The cells communicate using quorum sensing molecules LasI and RhlI, which are molecules that can diffuse between the cells.

The result of asynchronous logic synthesis is shown in **Figure 7.4**. This circuit is composed of three *generalized Muller C-elements* (gC), each of which is implemented as an independent circuit in a different cell. This gC behaves as follows: its output goes high when both inputs are high, and its output goes low only after the input not marked with a "+" goes low. In other words, if the input marked with a "+" goes low first, it remains high. The complete circuit behaves as follows. If the first circuit receives the Start signal (aTc) and the Sensor input (IPTG), it will produce a quorum sensing signal (LasI). If the Sensor input (IPTG) is still present, the second circuit would receive the first quorum sensing signal and the Sensor input, thus producing a second quorum sensing molecule (RhlI). The third circuit, if it receives this second quorum sensing signal and the Sensor input (IPTG) is still present, can produce the Actuator output (YFP). The purpose of this circuit is to work as a low-pass filter, as the Sensor input has to be present during the whole process in order to produce the Actuator. Otherwise, if the Sensor input disappears before the Actuator is produced, all the gC gates reset to low and LasI, RhlI, and YFP are no longer produced and are degraded away. This means that the circuit does not produce an Actuator output if the Sensor input is present only briefly, "filtering out" noise on the Sensor input.

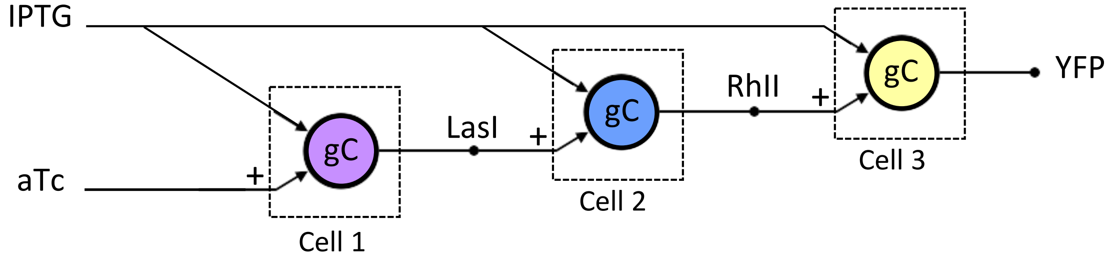


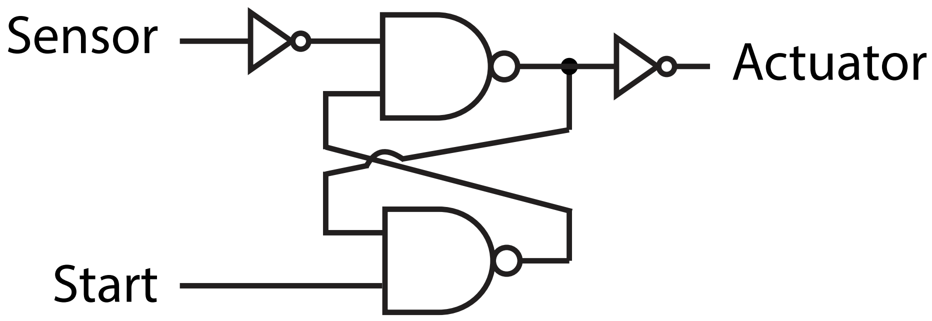
Figure 7.4: A genetic sensor that uses filtering and communication to improve its reliability. The logic diagram produced by logic synthesis. It is composed of three gC gates that go high when both inputs are high and go low when the input not marked with a "+" goes low. The output of the second and third gate are connected to the "+" input of the next gate. The detection begins when both IPTG and aTc go high, activating Cell 1. This creates the quorum signal LasI to diffuse to Cell 2, which then activates Cell 2 to produce the quorum signal RhlI. The RhlI signal diffuses to Cell 3 to activate YFP production. However, if IPTG goes low at any point during this chain reaction, the whole circuit resets.

7.3 Results of Our Technology Mapping Procedure

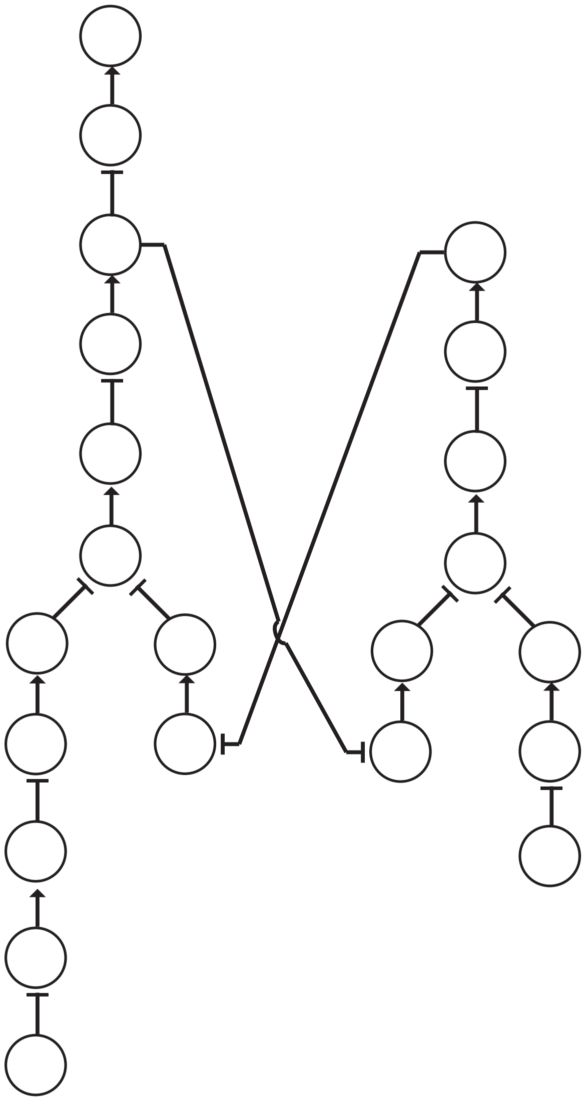
Before performing the technology mapping procedure, the sensor circuit in **Figure 7.2** and **Figure 7.3** are synthesized to a structural design shown in **Figure 7.5**. Then, this structural design is decomposed into *NAND* logic shown in **Figure 7.6**. **Figure 7.5** represents the designed circuit after running ATACS. **Figure 7.6** is a decomposed representation of the circuit using *NOT* and *NAND* logic after running YOSYS. The decomposed structural design is described using *NAND* logic because the CELLO library gates were generated in *NAND* and *NOT* gates. Then, technology mapping is performed on the decomposed gC to generate three different netlist solutions that can be used to assemble into a complete low-pass filter. Preselection is enabled when running technology mapping to ensure that the input signals going into the a gC circuit include small molecule signals. Specifically, the Sensor input signal is assigned to CELLO's LacI protein, Start is assigned to CELLO's TetR protein, and Actuator is assigned to CELLO's YFP protein. Then, the branch and bound algorithm is selected to perform technology mapping on the gC circuit. The result of technology mapping using IBIOSIM is shown in **Figure 7.7** for Cell 1. Sensors that are needed for initializing the input signals TetR and LacI proteins are manually added to the circuit for Cell 1. The technology mapping actually produces distinct genetic circuit sequences. Each sequence can be used to construct a plasmid that can be transformed into a separate culture of cells. As shown in **Figure 7.8**, the result would be three different cell types, which when mixed together form a population of cells that would implement the entire composite circuit. A filter model was then created in IBIOSIM's project workspace to connect three instances of the gC netlists in series. Quorum sensing molecules (LasI and RhlI) were manually added into the filter model for connecting the three gC circuits from

```
module Start_Sensor_Actuator_net(Start, Sensor, Actuator);

  input Start;
  input Sensor;
  output Actuator;

assign Actuator = (Start & Sensor) | (Sensor & Actuator);
endmodule
```

Figure 7.5: Structural Verilog of a generalized C-element after running ATACS.

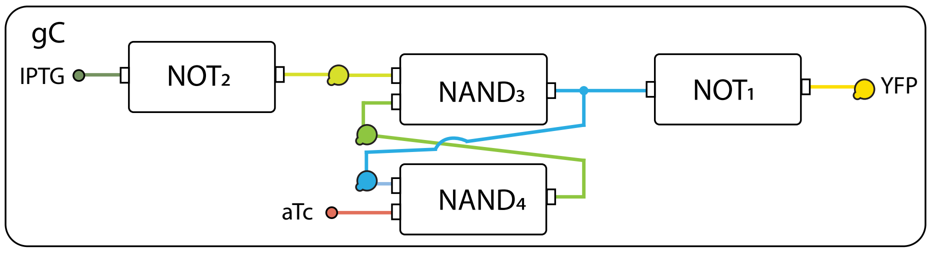


(a)

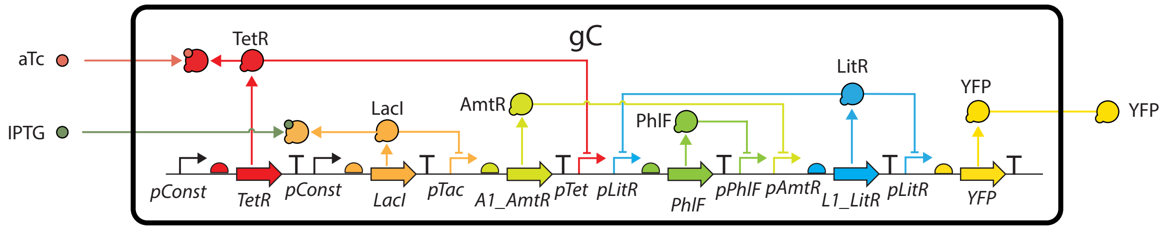


(b)

Figure 7.6: A decomposed gC described using *NOT* and *NAND* gates after running YOSYS. **Figure 7.6a** represents the decomposed circuit using logic gates. **Figure 7.6b** represents the decomposed circuit using *DecomposedGraph*.



(a)



(b)

Figure 7.7: Result of circuit after running technology mapping. **Figure 7.7a** represents a netlist of the genetic circuit. **Figure 7.7b** is a flattened representation of the netlist designed on a plasmid.

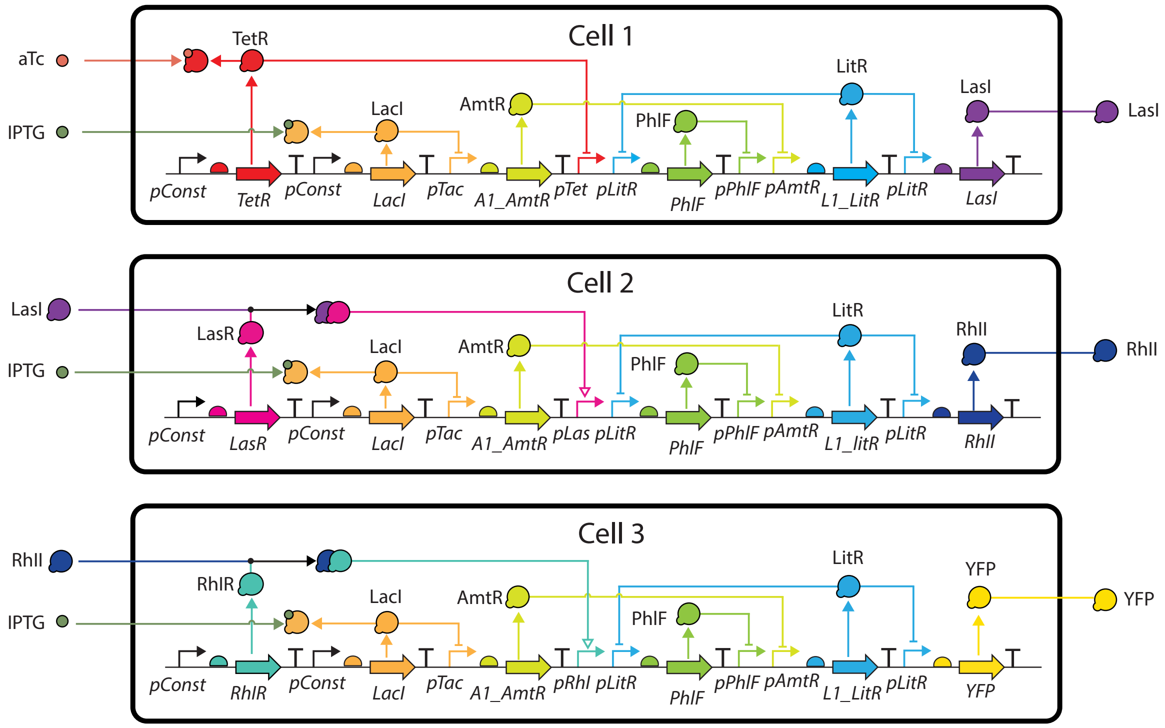


Figure 7.8: The genetic design produced by iBioSim, which is composed of three genetic sequences that can be put onto separate plasmids and transformed into cells to create three cell types.

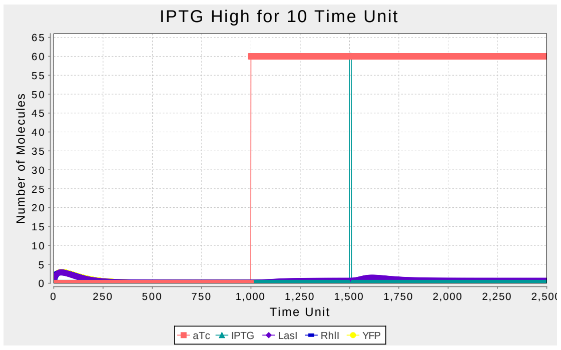
Cell 1, Cell 2, to Cell 3 because these molecules do not exist in the CELLO library. This completes the full design of a low-pass filter.

The complete circuit behaves as follows. If the first circuit receives the Start signal (aTc) and the Sensor input (IPTG), it will produce a quorum sensing signal (LasI). If the Sensor input (IPTG) is still present, the second circuit would receive the first quorum sensing signal and the Sensor input, thus producing a second quorum sensing molecule (RhlI). The third circuit, if it receives this second quorum sensing signal and the Sensor input (IPTG) is still present, can produce the Actuator output (YFP). The purpose of this circuit is to work as a low-pass filter, as the Sensor input has to be present during the whole process in order to produce the Actuator. Otherwise, if the Sensor input disappears before the Actuator is produced, all the gC gates reset to low and LasI, RhlI, and YFP are no longer produced and are degraded away. This means that the circuit does not produce an Actuator output if the Sensor input is present only briefly "filtering out" noise on the Sensor input.

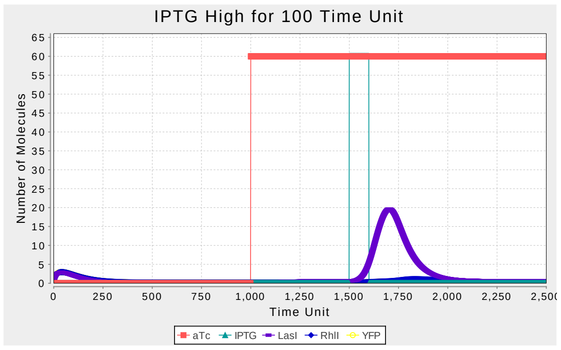
A testbench for the filter was created to generate a simulation to verify the behavior of the low-pass filter. Using iBIOSIM, ODE simulation of the circuit was performed, indicating the filtering behavior of the circuit shown in **Figure 7.9**. In each case, the circuit was exposed to varying amounts of times in which the signal IPTG is high, which produces different responses of the system as a whole (see **Figure 7.9b**, **Figure 7.9c**, and **Figure 7.9d**). In those cases where the exposure time to IPTG is not sufficient, the system acts as a filter and does not produce the YFP reporter protein (see **Figure 7.9a**). The YFP reporter protein is only produced when the system has been exposed to IPTG for sufficient time so that each subsystem (cell) produces its corresponding output before the signal IPTG is removed (see **Figure 7.9d**).

7.4 Discussion

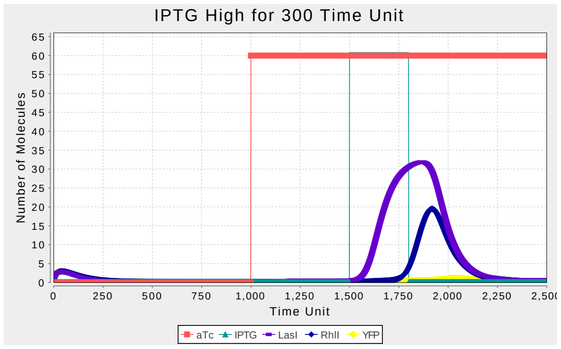
Separating the circuits into separate cells has multiple advantages. First, it reduces crosstalk problems and allows the same gates to be reused. Second, it increases the delay of the circuit to improve the filtering performance. Third, it enables further robustness as errors caused by hazards or other noise sources in a small number of cells can be tolerated, since the overall behavior is determined by the population dynamics. The decision to split the circuit across three cells is to create a delay to filter out momentary pulses of IPTG. In



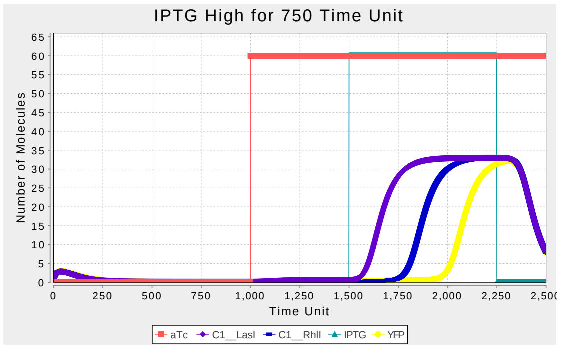
(a)



(b)



(c)



(d)

Figure 7.9: iBioSim ODE Runge-Kutta simulations (measured in number of molecules) of the genetic sensor demonstrating the filtering behavior. For the circuit outputs to turn on (LasI, RhlI, and YFP for the three cells, respectively), both aTc and IPTG inputs need to be present; however, if the IPTG signal is only briefly present, then the circuit filters this and does not show the YFP output. This behavior can be seen in the four different panels where the IPTG is present for differing amounts of time. **Figure 7.9a** IPTG is present for 10 time units and almost no LasI is produced. **Figure 7.9b** IPTG is present for 100 time units and LasI is produced by the first cell but only a little RhlI is produced by the second cell. **Figure 7.9c** IPTG is present for 300 time units seconds and both LasI and RhlI are produced with a clear delay between the two peaks. **Figure 7.9d** IPTG is present for 750 time units, providing sufficient time for the final circuit output (YFP) to be produced.

order to create a realistic design, however, we were limited to no more than three cells, since the number of known orthogonal quorum sensing molecules is limited (namely, LasI and RhlI). Even limiting to three cell types, there are additional challenges in engineering microbial consortia. Some of these challenges are shared with engineering of homogeneous cell populations. Others, such as maintaining stable cell proportions, avoiding horizontal gene-transfer [159], or engineering stable cell-cell communications [160], are specific for multicellular engineering and require special attention [161].

CHAPTER 8

CONCLUSION

Ground breaking applications have been developed in the past couple of years due to the rate at which the synthetic biology field is growing. Promising efforts have been aided by adapting engineering principles that are used for designing, building, and testing genetic circuits. Such efforts have resulted in the development of many applications built from genetic circuits composed of combinational logic gates and memory gates. While there are many GDA tools that exist for assembling combinational logic gates, there are few GDA tools that have been designed for sequential circuits.

This dissertation demonstrates a workflow to construct asynchronous genetic circuits by leveraging asynchronous logic design methods. Asynchronous design is selected for designing genetic circuits in this workflow because it best reflects how cellular biology behaves on a molecular scale rather than that of a synchronous design style. Biological data standards, such as SBOL and SBML, play an important role for GDA tools and workflows. Data standards are essential to ensure that information remains consistent when it is used for different purposes and used across different tools. Data standards also ensure that the contents recorded in these data formats are reproducible for sharing and publishing. This chapter highlights the main contributions of this dissertation, summarized in Section 8.1, and discusses future directions of this research in Section 8.2.

8.1 Summary

A key design strategy for electronic circuits is to use abstraction to separate the behavioral description from the physical design for a targeted circuit. A behavioral design is translated into physical components through several steps and verification procedures are used to ensure correctness at each step. Doing so allows circuits to be designed at a larger scale and develop more complex functionalities.

This dissertation presents an asynchronous circuit design workflow for genetic circuits.

The proposed workflow follows many electronic circuit design methodologies, such as the use of HDLs to describe behavioral designs. More specifically, Verilog is used in this workflow for designing genetic circuits. A Verilog compiler was created for this workflow to translate behavioral Verilog to structural Verilog. A subset of the Verilog language is supported in this Verilog compiler for designing genetic circuits in an asynchronous style. Using the created Verilog compiler, the workflow starts with a behavioral Verilog design to describe the design specification and a testbench to verify the specification's behavior. The behavioral Verilog design is then compiled into an LPN that is fed into ATACS to perform hazard-free asynchronous logic synthesis. ATACS synthesizes the LPN design into a structural logic design in Verilog. The resulting structural Verilog design can then be imported into iBioSim for technology mapping to produce a physical design. A library of genetic gates must be provided to the technology mapping procedure in order to generate a netlist of genetic gates that fits the structural Verilog description. An automated procedure to construct genetic gates from a list of transcriptional units was created for this workflow to construct a large library of genetic gates. A gate identification method is also supported to categorize genetic gates into its corresponding logic. Genetic gates exhibiting the same logic behavior but that are structurally different can be identified in this process. Gates that are not identified can be included in the library of genetic gates by providing a Verilog file describing the gate's behavior. Technology mapping can proceed once a library of genetic gates and a design specification are provided. The technology mapping procedure discussed in this dissertation leverages the same existing technology mapping functionality supported in iBioSim. A key difference is that the proposed technology mapping procedure can handle feedback loops that are found in sequential circuits. The output of this technology mapping procedure produces a netlist of genetic gates described in the SBOL format. Verification is performed to validate that the behavior of the physical design is consistent with the original Verilog design. Case studies are discussed in this dissertation and demonstrate how the proposed workflow can be applied to design complex genetic circuits programatically.

8.2 Future Work

While the proposed workflow is promising, there is room for improvements. This section goes over areas in the proposed workflow that could be expanded.

8.2.1 ATACS Search Space

ATACS, currently, does not ensure that all search spaces are explored for the input files that are provided. Take the testbench for an SR latch as an example. ATACS exploration depends on the contents of the testbench. If the provided testbench does not cover all the corner cases that might be important for testing the designed circuit, ATACS might not produce any valid solution. If ATACS cannot generate a solution, then errors are reported back after running the tool that can be difficult to comprehend for those without domain expertise. A future work needed in this area is to add more support to ATACS to have a better mechanism to evaluate coverage to ensure that a complete search space has been performed based on the input files that are provided.

8.2.2 Replicating ATACS for Genetic Circuits

ATACS was originally built for synthesizing asynchronous electronic circuit design. As a result, this synthesizing tool accounts for hazards that could occur in asynchronous electronic circuit design that does not apply to asynchronous genetic circuit design. An area where this research can expand on is to replicate this tool and modify the requirements so that the constraints considered when performing synthesis are better suited for genetic circuit design.

8.2.3 Curate Part Library into SBOL

In order to create larger gate libraries, more parts need to be collected. There are several databases that contains useful information for building multiple libraries of genetic gates and they include information for bacterias, plants, and fungi, among others, that could be used to build genetic circuits. However, curating these databases into SBOL takes effort for a couple of reasons. First, not all databases have all core information that is needed for building genetic gates. For example, iGEM has a public database with a large collection of parts that can be used for building gates but information about interactions is not provided. If information about interactions is missing, then VPR will not be able to mine information that can be used for building gates. The type of information that can be built in this case is limited to transcriptional units. Second, databases are stored in varying data formats. Converting information from different databases into SBOL requires knowledge about the different data representations.

8.2.4 Evaluate Promoter Location to Assemble Transcriptional Units

It is known that the position of the promoters on a transcriptional unit affects the transcription of DNA. Promoters that are selected in this workflow does not account for its relative position to where the transcriptional start site occurs. This area can be expanded to provide a better insight when selecting DNA parts to build transcriptional units. The SBOL data format has this information already supported and available to access using the *SequenceAnnotation* data object. In addition, the CELLO dataset has this location property provided for all promoter parts that are used within this workflow when assembling transcriptional units. Similar to how roadblock is addressed when building transcriptional units, selecting promoters can be considered as an extra constraint when assembling transcriptional units.

8.2.5 Support Additional Genetic Gate Types

Based on literature, there are more types of logic families that can be supported during the gate identification procedure presented in Section 4.3. Some of these gate types include *XOR*, *XNOR*, *BUFFER*, and C-element gates. However, the construction of these gates must be perused from literature in order to understand how to represent these gates. While these gates such as *XOR* and *XNOR* can be composed from gates that are already supported in this workflow, the structure of these gates can take on different forms.

Currently, *NOTSUPPORTED* gates can be used to define gates that are not recognized in the gate identification step. Recall from Section 4.3 that using *NOTSUPPORTED* gates in technology mapping is done by supplying a Verilog file that defines the behavior of the gate. However, using *NOTSUPPORTED* gate is not a sufficient approach if a genetic gate can be defined within the gate identification process. For example, if VPR generates C-element gates that are not identified, then these gates will need to be manually identified from the list of *NOTSUPPORTED* gates. Providing the technology mapping procedure one C-element in the library of genetic gate is not tedious because only one Verilog file is needed to describing this gate's behavior. However, having more variants of the C-element gate can be a problem because a Verilog file would need to be provided for each variant, which can be a tedious task.

Lastly, another possible way to support additional gate types is to consider gate families other than transcriptional regulation gates, such as recombinase, CRISPR/CAS9, etc.

8.2.6 Expand Signal Mismatch

The only SYNBIOHUB collection that has information about response functions to address signal mismatch is the CELLO *E. Coli* dataset. Response functions are used on genetic gates that are built on a PoP-style method. Genetic gates, using the PoP-style method, are defined as the effect of the rate of transcripts of an upstream CDS on the rate of transcripts produced by a downstream promoter that it regulates. This workflow interprets input and output signal for a genetic gate based on signal molecule counts. The technology mapping procedure will need to be modified in order to use the response function provided in the CELLO dataset.

8.2.7 Evaluating Threshold of Genetic Gates Before Cell Death

Cells cannot be overloaded with genetic gates. If there are too many genetic parts within the cell, this can cause the cell to potentially die. This area of work could be better understood by characterizing the burden used on a cell for a genetic circuit. For example, the CELLO data used in this research have toxicity data for the parts used to assemble the genetic gates. This toxicity data could be used to replace the cost of sequence length when selecting gates in the technology mapping procedure. The circuit can then use these toxicity data to model the threshold where cell death occurs.

8.2.8 Increase Technology Mapping Performance

The runtime when calling the technology mapping procedure is slow for a large library and on a big specification. There is a need to explore a faster way to perform the technology mapping procedure. Lehman et al. [162] have proposed a solution that decreases the runtime down to a logarithmic complexity that is proportional to the size of the specification's graph.

8.2.9 Reduce Structural Bias in Technology Mapping

The genetic gates supported in this workflow are 1-input gates and 2-input gates. When these gates are decomposed, the structures are symmetrical. However, if the gates included in the library go beyond 2-input gates, then the decomposed structure for these gates will affect how they are paired to the specification. Another area of work that this research can expand on is to support multiple decomposition structure for large input gates in order to prevent biasing of structure when pairing genetic gates to the specification.

8.2.10 Enriched Parts to Improve Genetic Circuit Models

Default parameter values are currently used when modeling genetic circuits from the technology mapping procedure. These default parameter, when used for simulating an SBML model of a genetic circuit, produces a generic simulation result. The output values and the prediction of hazards are calculated from these default parameters. There is a need to support parameters that are specific to the part used in the circuit. Retrieving these parameter values is accomplished by adding a layer of characterization data on the SBOL designs produced from the technology mapping tool. These characterization data will be accessed by the VPR model generation procedure described by Misirili et al [149]. This will, ideally, provide a more accurate depiction of the biological parts that were selected for the technology mapping procedure and how well it behaves in the circuit.

REFERENCES

[1] M. F. Wolff, “The secret six-month project: why Texas Instruments decided to put the first transistor radio on the market by Christmas 1954 and how it was accomplished,” *IEEE Spectrum*, vol. 22, no. 12, pp. 64–69, 1985.

[2] D. Thomas and P. Moorby, *The Verilog® Hardware Description Language*. Springer Science & Business Media, 2008.

[3] P. J. Ashenden, *The Designer's Guide to VHDL*. Morgan Kaufmann, 2010, vol. 3.

[4] D. E. Cameron, C. J. Bashor, and J. J. Collins, “A brief history of synthetic biology,” *Nature Reviews Microbiology*, vol. 12, no. 5, p. 381, 2014.

[5] T. S. Gardner, C. R. Cantor, and J. J. Collins, “Construction of a genetic toggle switch in escherichia coli,” *Nature*, vol. 403, no. 6767, p. 339, 2000.

[6] M. B. Elowitz and S. Leibler, “A synthetic oscillatory network of transcriptional regulators,” *Nature*, vol. 403, no. 6767, p. 335, 2000.

[7] C. C. Guet, M. B. Elowitz, W. Hsing, and S. Leibler, “Combinatorial synthesis of genetic networks,” *Science*, vol. 296, no. 5572, pp. 1466–1470, 2002.

[8] I. Hoteit, N. Kharma, and L. Varin, “Computational simulation of a gene regulatory network implementing an extendable synchronous single-input delay flip-flop,” *BioSystems*, vol. 109, no. 1, pp. 57–71, 2012.

[9] J. Stricker, S. Cookson, M. R. Bennett, W. H. Mather, L. S. Tsimring, and J. Hasty, “A fast, robust and tunable synthetic gene oscillator,” *Nature*, vol. 456, no. 7221, p. 516, 2008.

[10] G. Rodrigo and A. Jaramillo, “Computational design of digital and memory biological devices,” *Systems and Synthetic Biology*, vol. 1, no. 4, p. 183, 2007.

[11] J. Sardanyés, A. Bonforti, N. Conde, R. Solé, and J. Macia, “Computational implementation of a tunable multicellular memory circuit for engineered eukaryotic consortia,” *Frontiers in Physiology*, vol. 6, 2015.

[12] A. Urrios, J. Macia, R. Manzoni, N. Conde, A. Bonforti, E. de Nadal, F. Posas, and R. SolÕẩ, “A synthetic multicellular memory device,” *ACS Synthetic Biology*, vol. 5, no. 8, pp. 862–873, 2016.

[13] L. Andrews, A. Nielsen, and C. Voigt, “Cellular checkpoint control using programmable sequential logic,” *Science*, vol. 361, no. 6408, 2018. [Online]. Available: http://science.sciencemag.org/content/361/6408/eaap8987

[14] B. H. Weinberg, N. H. Pham, L. D. Caraballo, T. Lozanoski, A. Engel, S. Bhatia, and W. W. Wong, “Large-scale design of robust genetic circuits with multiple inputs and outputs for mammalian cells,” *Nature Biotechnology*, vol. 35, no. 5, p. 453, 2017.

[15] A. A. K. Nielsen, B. S. Der, J. Shin, P. Vaidyanathan, V. Paralanov, E. A. Strychalski, D. Ross, D. Densmore, and C. A. Voigt, "Genetic circuit design automation," *Science*, vol. 352, no. 6281, 2016. [Online]. Available: http://science.sciencemag.org/content/352/6281/aac7341

[16] T. S. Moon, C. Lou, A. Tamsir, B. C. Stanton, and C. A. Voigt, "Genetic programs constructed from layered logic gates in single cells," *Nature*, vol. 491, no. 7423, p. 249, 2012.

[17] G. Schendzielorz, M. Dippong, A. GrÕìnberger, D. Kohlheyer, A. Yoshida, S. Binder, C. Nishiyama, M. Nishiyama, M. Bott, and L. Eggeling, "Taking control over control: use of product sensing in single cells to remove flux control at key enzymes in biosynthesis pathways," *ACS Synthetic Biology*, vol. 3, no. 1, pp. 21–29, 2013.

[18] 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," *Nature Biotechnology*, vol. 30, no. 4, pp. 354–359, 2012.

[19] T.-M. Yi, Y. Huang, M. I. Simon, and J. Doyle, "Robust perfect adaptation in bacterial chemotaxis through integral feedback control," *Proceedings of the National Academy of Sciences*, vol. 97, no. 9, pp. 4649–4653, 2000.

[20] K. Krishnanathan, S. R. Anderson, S. A. Billings, and V. Kadirkamanathan, "A data-driven framework for identifying nonlinear dynamic models of genetic parts," *ACS Synthetic Biology*, vol. 1, no. 8, pp. 375–384, 2012.

[21] P. Carbonell, P. Parutto, C. Baudier, C. Junot, and J.-L. Faulon, "Retropath: automated pipeline for embedded metabolic circuits," *ACS Synthetic Biology*, vol. 3, no. 8, pp. 565–577, 2013.

[22] B. L. Adams, K. K. Carter, M. Guo, H.-C. Wu, C.-Y. Tsao, H. O. Sintim, J. J. Valdes, and W. E. Bentley, "Evolved quorum sensing regulator, LsrR, for altered switching functions," *ACS Synthetic Biology*, vol. 3, no. 4, pp. 210–219, 2013.

[23] T. Umeyama, S. Okada, and T. Ito, "Synthetic gene circuit-mediated monitoring of endogenous metabolites: identification of GAL11 as a novel multicopy enhancer of S-adenosylmethionine level in yeast," *ACS Synthetic Biology*, vol. 2, no. 8, pp. 425–430, 2013.

[24] J. A. Stapleton, K. Endo, Y. Fujita, K. Hayashi, M. Takinoue, H. Saito, and T. Inoue, "Feedback control of protein expression in mammalian cells by tunable synthetic translational inhibition," *ACS Synthetic Biology*, vol. 1, no. 3, pp. 83–88, 2011.

[25] M. H. Medema, R. Breitling, R. Bovenberg, and E. Takano, "Exploiting plug-and-play synthetic biology for drug discovery and production in microorganisms," *Nature Reviews. Microbiology*, vol. 9, no. 2, p. 131, 2011.

[26] M. Fischbach and C. A. Voigt, "Prokaryotic gene clusters: a rich toolbox for synthetic biology," *Biotechnology Journal*, vol. 5, no. 12, pp. 1277–1296, 2010.

[27] H.-J. Frasch, M. H. Medema, E. Takano, and R. Breitling, "Design-based re-engineering of biosynthetic gene clusters: plug-and-play in practice," *Current Opinion in Biotechnology*, vol. 24, no. 6, pp. 1144–1150, 2013.

[28] K. Temme, D. Zhao, and C. A. Voigt, "Refactoring the nitrogen fixation gene cluster from klebsiella oxytoca," *Proceedings of the National Academy of Sciences*, vol. 109, no. 18, pp. 7085–7090, 2012.

[29] Z. Shao, G. Rao, C. Li, Z. Abil, Y. Luo, and H. Zhao, "Refactoring the silent spectinabilin gene cluster using a plug-and-play scaffold," *ACS Synthetic Biology*, vol. 2, no. 11, pp. 662–669, 2013.

[30] C. Oßwald, G. Zipf, G. Schmidt, J. Maier, H. S. Bernauer, R. MÕiller, and S. C. Wenzel, "Modular construction of a functional artificial epothilone polyketide pathway," *ACS Synthetic Biology*, vol. 3, no. 10, pp. 759–772, 2012.

[31] L. Steidler, W. Hans, L. Schotte, S. Neirynck, F. Obermeier, W. Falk, W. Fiers, and E. Remaut, "Treatment of murine colitis by lactococcus lactis secreting interleukin-10," *Science*, vol. 289, no. 5483, pp. 1352–1355, 2000.

[32] J. C. Anderson, E. J. Clarke, A. P. Arkin, and C. A. Voigt, "Environmentally controlled invasion of cancer cells by engineered bacteria," *Journal of Molecular Biology*, vol. 355, no. 4, pp. 619–627, 2006.

[33] W. C. Ruder, T. Lu, and J. J. Collins, "Synthetic biology moving into the clinic," *Science*, vol. 333, no. 6047, pp. 1248–1252, 2011.

[34] J.-P. Motta, L. G. Bermúdez-Humarán, C. Deraison, L. Martin, C. Rolland, P. Rousset, J. Boue, G. Dietrich, K. Chapman, P. Kharrat *et al.*, "Food-grade bacteria expressing elafin protect against inflammation and restore colon homeostasis," *Science Translational Medicine*, vol. 4, no. 158, pp. 158ra144–158ra144, 2012.

[35] S. Wang, Q. Kong, and R. Curtiss, "New technologies in developing recombinant attenuated salmonella vaccine vectors," *Microbial Pathogenesis*, vol. 58, pp. 17–28, 2013.

[36] J. H. Huh, J. T. Kittleson, A. P. Arkin, and J. C. Anderson, "Modular design of a synthetic payload delivery device," *ACS Synthetic Biology*, vol. 2, no. 8, pp. 418–424, 2013.

[37] S. Gupta, E. E. Bram, and R. Weiss, "Genetically programmable pathogen sense and destroy," *ACS Synthetic Biology*, vol. 2, no. 12, pp. 715–723, 2013.

[38] I. Y. Hwang, M. H. Tan, E. Koh, C. L. Ho, C. L. Poh, and M. W. Chang, "Reprogramming microbes to be pathogen-seeking killers," *ACS Synthetic Biology*, vol. 3, no. 4, pp. 228–237, 2013.

[39] A. Prindle, J. Selimkhanov, T. Danino, P. Samayoa, A. Goldberg, S. N. Bhatia, and J. Hasty, "Genetic circuits in salmonella typhimurium," *ACS Synthetic Biology*, vol. 1, no. 10, pp. 458–464, 2012.

[40] K. Volzing, J. Borrero, M. J. Sadowsky, and Y. N. Kaznessis, "Antimicrobial peptides targeting Gram-negative pathogens, produced and delivered by lactic acid bacteria," *ACS Synthetic Biology*, vol. 2, no. 11, pp. 643–650, 2013.

[41] J. Hasty, "Engineered microbes for therapeutic applications," *ACS Synthetic Biology*, vol. 1, no. 10, pp. 438–439, 2012.

[42] T. Danino, J. Lo, A. Prindle, J. Hasty, and S. N. Bhatia, "In vivo gene expression dynamics of tumor-targeted bacteria," *ACS Synthetic Biology*, vol. 1, no. 10, pp. 465–470, 2012.

[43] E. J. Archer, A. B. Robinson, and G. M. SÕìel, "Engineered e. coli that detect and respond to gut inflammation through nitric oxide sensing," *ACS Synthetic Biology*, vol. 1, no. 10, pp. 451–457, 2012, pMID: 23656184. [Online]. Available: https://doi.org/10.1021/sb3000595

[44] M. S. Antunes, K. J. Morey, J. J. Smith, K. D. Albrecht, T. A. Bowen, J. K. Zdunek, J. F. Troupe, M. J. Cuneo, C. T. Webb, H. W. Hellinga *et al.*, "Programmable ligand detection system in plants through a synthetic signal transduction pathway," *PLoS One*, vol. 6, no. 1, p. e16292, 2011.

[45] D. M. Widmaier, D. Tullman-Ercek, E. A. Mirsky, R. Hill, S. Govindarajan, J. Minshull, and C. A. Voigt, "Engineering the salmonella type iii secretion system to export spider silk monomers," *Molecular Systems Biology*, vol. 5, no. 1, p. 309, 2009.

[46] K. Bernhardt, N. S. Chand, E. Carter, J. Lee, Y. Xu, X. Zhu, D. Rowe, J. W. Ajioka, J. Goncalves, J. Haseloff *et al.*, "New tools for self-organized pattern formation," *BMC Systems Biology*, vol. 1, no. 1, p. S10, 2007.

[47] X.-X. Xia, Z.-G. Qian, C. S. Ki, Y. H. Park, D. L. Kaplan, and S. Y. Lee, "Native-sized recombinant spider silk protein produced in metabolically engineered escherichia coli results in a strong fiber," *Proceedings of the National Academy of Sciences*, vol. 107, no. 32, pp. 14 059–14 063, 2010.

[48] D. M. Widmaier and C. A. Voigt, "Quantification of the physiochemical constraints on the export of spider silk proteins by salmonella type iii secretion," *Microbial Cell Factories*, vol. 9, no. 1, p. 78, 2010.

[49] F. Aquea, F. Federici, C. Moscoso, A. Vega, P. Jullian, J. Haseloff, and P. ARCE-JOHNSON, "A molecular framework for the inhibition of arabidopsis root growth in response to boron toxicity," *Plant, Cell & Environment*, vol. 35, no. 4, pp. 719–734, 2012.

[50] M. S. Antunes, S.-B. Ha, N. Tewari-Singh, K. J. Morey, A. M. Trofka, P. Kugrens, M. Deyholos, and J. I. Medford, "A synthetic de-greening gene circuit provides a reporting system that is remotely detectable and has a re-set capacity," *Plant Biotechnology Journal*, vol. 4, no. 6, pp. 605–622, 2006.

[51] E. Oberortner and D. Densmore, "Web-based software tool for constraint-based design specification of synthetic biological systems," *ACS Synthetic Biology*, vol. 4, no. 6, pp. 757–760, 2014.

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

[53] J. Beal, T. Lu, and R. Weiss, "Automatic compilation from high-level biologically-oriented programming language to genetic regulatory networks," *PloS One*, vol. 6, no. 8, p. e22490, 2011.

[54] E. H. Wilson, S. Sagawa, J. W. Weis, M. G. Schubert, M. Bissell, B. Hawthorne, C. D. Reeves, J. Dean, and D. Platt, "Genotype specification language," *ACS Synthetic Biology*, vol. 5, no. 6, pp. 471–478, 2016.

[55] L. P. Smith, F. T. Bergmann, D. Chandran, and H. M. Sauro, "Antimony: a modular model definition language," *Bioinformatics*, vol. 25, no. 18, pp. 2452–2454, 2009.

[56] 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.

[57] G. Misirli, J. S. Hallinan, T. Yu, J. R. Lawson, S. M. Wimalaratne, M. T. Cooling, and A. Wipat, "Model annotation for synthetic biology: automating model to nucleotide sequence conversion," *Bioinformatics*, vol. 27, no. 7, pp. 973–979, 2011.

[58] G. Rodrigo, J. Carrera, and A. Jaramillo, "Asmparts: assembly of biological model parts," *Systems and Synthetic Biology*, vol. 1, no. 4, pp. 167–170, 2007.

[59] M. Quintin, N. J. Ma, S. Ahmed, S. Bhatia, A. Lewis, F. J. Isaacs, and D. Densmore, "Merlin: computer-aided oligonucleotide design for large scale genome engineering with MAGE," *ACS Synthetic Biology*, vol. 5, no. 6, pp. 452–458, 2016.

[60] N. Roehner, E. M. Young, C. A. Voigt, D. B. Gordon, and D. Densmore, "Double Dutch: a tool for designing combinatorial libraries of biological systems," *ACS Synthetic Biology*, vol. 5, no. 6, pp. 507–517, 2016.

[61] H. M. Salis, E. A. Mirsky, and C. A. Voigt, "Automated design of synthetic ribosome binding sites to control protein expression," *Nature Biotechnology*, vol. 27, no. 10, p. 946, 2009.

[62] A. Espah Borujeni, A. S. Channarasappa, and H. M. Salis, "Translation rate is controlled by coupled trade-offs between site accessibility, selective RNA unfolding and sliding at upstream standby sites," *Nucleic Acids Research*, vol. 42, no. 4, pp. 2646–2659, 2013.

[63] M. J. Czar, Y. Cai, and J. Peccoud, "Writing DNA with GenoCAD," *Nucleic Acids Research*, vol. 37, no. suppl_2, pp. W40–W47, 2009.

[64] L. Huynh, A. Tsoukalas, M. Köppe, and I. Tagkopoulos, "SBROME: a scalable optimization and module matching framework for automated biosystems design," *ACS Synthetic Biology*, vol. 2, no. 5, pp. 263–273, 2013.

[65] A. Espah Borujeni and H. M. Salis, "Translation initiation is controlled by RNA folding kinetics via a ribosome drafting mechanism," *Journal of the American Chemical Society*, vol. 138, no. 22, pp. 7016–7023, 2016.

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

[67] M. Zuker, "Mfold web server for nucleic acid folding and hybridization prediction," *Nucleic Acids Research*, vol. 31, no. 13, pp. 3406–3415, 2003.

[68] A. Leaver-Fay, M. Tyka, S. M. Lewis, O. F. Lange, J. Thompson, R. Jacak, K. W. Kaufman, P. D. Renfrew, C. A. Smith, W. Sheffler *et al.*, "ROSETTA3: an object-oriented software suite for the simulation and design of macromolecules," in *Methods in Enzymology*. Elsevier, 2011, vol. 487, pp. 545–574.

[69] J. T. Bates, D. Chivian, and A. P. Arkin, "GLAMM: genome-linked application for metabolic maps," *Nucleic Acids Research*, vol. 39, no. suppl_2, pp. W400–W405, 2011.

[70] M. S. Dasika and C. D. Maranas, "OptCircuit: an optimization based method for computational design of genetic circuits," *BMC Systems Biology*, vol. 2, no. 1, p. 24, 2008.

[71] G. Rodrigo and A. Jaramillo, "AutoBioCAD: full biodesign automation of genetic circuits," *ACS Synthetic Biology*, vol. 2, no. 5, pp. 230–236, 2012.

[72] M. A. Marchisio, "Parts & pools: a framework for modular design of synthetic gene circuits," *Frontiers in Bioengineering and Biotechnology*, vol. 2, p. 42, 2014.

[73] V. Vasilev, C. Liu, T. Haddock, S. Bhatia, A. Adler, F. Yaman, J. Beal, J. Babb, R. Weiss, D. Densmore *et al.*, "A software stack for specification and robotic execution of protocols for synthetic biological engineering," in *3rd International Workshop on Bio-Design Automation*. Citeseer, 2011.

[74] E. Appleton, J. Tao, T. Haddock, and D. Densmore, "Interactive assembly algorithms for molecular cloning," *Nature Methods*, vol. 11, no. 6, p. 657, 2014.

[75] J. Blakes, O. Raz, U. Feige, J. Bacardit, P. Widera, T. Ben-Yehezkel, E. Shapiro, and N. Krasnogor, "Heuristic for maximizing DNA reuse in synthetic DNA library assembly," *ACS Synthetic Biology*, vol. 3, no. 8, pp. 529–542, 2014.

[76] H. Huang and D. Densmore, "Fluigi: microfluidic device synthesis for synthetic biology," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 11, no. 3, p. 26, 2014.

[77] N. J. Hillson, R. D. Rosengarten, and J. D. Keasling, "J5 DNA assembly design automation software," *ACS Synthetic Biology*, vol. 1, no. 1, pp. 14–21, 2011.

[78] G. Linshiz, N. Stawski, G. Goyal, C. Bi, S. Poust, M. Sharma, V. Mutalik, J. D. Keasling, and N. J. Hillson, "PR-PR: cross-platform laboratory automation system," *ACS Synthetic Biology*, vol. 3, no. 8, pp. 515–524, 2014.

[79] T. Koressaar and M. Remm, "Enhancements and modifications of primer design program Primer3," *Bioinformatics*, vol. 23, no. 10, pp. 1289–1291, 2007.

[80] A. Untergasser, I. Cutcutache, T. Koressaar, J. Ye, B. C. Faircloth, M. Remm, and S. G. Rozen, "Primer3-new capabilities and interfaces," *Nucleic Acids Research*, vol. 40, no. 15, pp. e115–e115, 2012.

[81] W. Huber, V. J. Carey, R. Gentleman, S. Anders, M. Carlson, B. S. Carvalho, H. C. Bravo, S. Davis, L. Gatto, T. Girke *et al.*, "Orchestrating high-throughput genomic analysis with bioconductor," *Nature Methods*, vol. 12, no. 2, p. 115, 2015.

[82] S. M. Castillo-Hair, J. T. Sexton, B. P. Landry, E. J. Olson, O. A. Igoshin, and J. J. Tabor, "FlowCal: a user-friendly, open source software tool for automatically converting flow cytometry data from arbitrary to calibrated units," *ACS Synthetic Biology*, vol. 5, no. 7, pp. 774–780, 2016.

[83] 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 Research*, vol. 40, no. 18, pp. e141–e141, 2012.

[84] C. Madsen, J. A. McLaughlin, G. Mısırlı, M. Pocock, K. Flanagan, J. Hallinan, and A. Wipat, "The SBOL stack: a platform for storing, publishing, and sharing synthetic biology designs," *ACS Synthetic Biology*, vol. 5, no. 6, pp. 487–497, 2016.

[85] D. Densmore, A. Van Devender, M. Johnson, and N. Sritanyaratana, "A platform-based design environment for synthetic biological systems," in *The Fifth Richard Tapia Celebration of Diversity in Computing Conference: Intellect, Initiatives, Insight, and Innovations*. ACM, 2009, pp. 24–29.

[86] E. Appleton, J. Tao, F. C. Wheatley, D. H. Desai, T. M. Lozanoski, P. D. Shah, J. A. Awtry, S. S. Jin, T. L. Haddock, and D. M. Densmore, "Owl: electronic datasheet generator," *ACS Synthetic Biology*, vol. 3, no. 12, pp. 966–968, 2014.

[87] V. Chelliah, C. Laibe, and N. Le Novère, "BioModels database: a repository of mathematical models of biological processes," in *Encyclopedia of Systems Biology*. Springer, 2013, pp. 134–138.

[88] C. J. Myers, N. A. Barker, K. R. Jones, H. Kuwahara, C. Madsen, and N.-P. D. Nguyen, "IBioSim: a tool for the analysis and design of genetic circuits." *Bioinformatics*, vol. 25, no. 21, pp. 2848–2849, 2009.

[89] L. Watanabe, T. Nguyen, M. Zhang, Z. Zundel, Z. Zhang, C. Madsen, N. Roehner, and C. Myers, "IBioSim 3: a tool for model-based genetic circuit design," *ACS Synthetic Biology*, vol. 8, no. 7, pp. 1560–1563, 2018.

[90] S. Hoops, S. Sahle, R. Gauges, C. Lee, J. Pahle, N. Simus, M. Singhal, L. Xu, P. Mendes, and U. Kummer, "COPASI: a complex pathway simulator," *Bioinformatics*, vol. 22, no. 24, pp. 3067–3074, 2006.

[91] A. D. Hill, J. R. Tomshine, E. M. Weeding, V. Sotiropoulos, and Y. N. Kaznessis, "SynBioSS: the synthetic biology modeling suite," *Bioinformatics*, vol. 24, no. 21, pp. 2551–2553, 2008.

[92] D. Chandran, F. T. Bergmann, and H. M. Sauro, "TinkerCell: modular CAD tool for synthetic biology," *Journal of Biological Engineering*, vol. 3, no. 1, p. 19, 2009.

[93] S. S. Jang, K. T. Oishi, R. G. Egbert, and E. Klavins, "Specification and simulation of synthetic multicelled behaviors," *ACS Synthetic Biology*, vol. 1, no. 8, pp. 365–374, 2012.

[94] J. Starruß, W. de Back, L. Brusch, and A. Deutsch, "Morpheus: a user-friendly modeling environment for multiscale and multicellular systems biology," *Bioinformatics*, vol. 30, no. 9, pp. 1331–1332, 2014.

[95] B. G. Olivier, J. M. Rohwer, and J.-H. S. Hofmeyr, "Modelling cellular systems with PySCeS," *Bioinformatics*, vol. 21, no. 4, pp. 560–561, 2005.

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

[97] H. Sauro, "Jarnac: an interactive metabolic systems language in computation in cells," in *Proceedings of an EPSRC Emerging Computing Paradigms Workshop*. Dept. of Computer Science Technical Report No. 345, University of Hertfordshire, 2000.

[98] C. Madsen, F. Shmarov, and P. Zuliani, "BioPSy: an SMT-based tool for guaranteed parameter set synthesis of biological models," in *International Conference on Computational Methods in Systems Biology*. Springer, 2015, pp. 182–194.

[99] E. T. Somogyi, J.-M. Bouteiller, J. A. Glazier, M. König, J. K. Medley, M. H. Swat, and H. M. Sauro, "LibRoadRunner: a high performance sbml simulation and analysis library," *Bioinformatics*, vol. 31, no. 20, pp. 3315–3321, 2015.

[100] S. Chakrabarti, C. J. Lanczycki, A. R. Panchenko, T. M. Przytycka, P. A. Thiessen, and S. H. Bryant, "Refining multiple sequence alignments with conserved core regions," *Nucleic Acids Research*, vol. 34, no. 9, pp. 2598–2606, 2006.

[101] 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 Expression and Purification*, vol. 47, no. 2, pp. 441–445, 2006.

[102] M. Zhang, J. A. McLaughlin, A. Wipat, and C. J. Myers, "SBOLDesigner 2: an intuitive tool for structural genetic design," *ACS Synthetic Biology*, vol. 6, no. 7, pp. 1150–1160, 2017.

[103] N. Roehner and C. J. Myers, "Directed acyclic graph-based technology mapping of genetic circuit models," *ACS Synthetic Biology*, vol. 3, no. 8, pp. 543–555, 2014.

[104] H. Baig and J. Madsen, "A top-down approach to genetic circuit synthesis and optimized technology mapping," in *9th International Workshop on Bio-Design Automation*, 2017.

[105] M. Madec, F. Pecheux, Y. Gendrault, E. Rosati, C. Lallement, and J. Haiech, "GeNeDA: An open-source workflow for design automation of gene regulatory networks inspired from microelectronics," *Journal of Computational Biology*, vol. 23, no. 10, pp. 841–855, 2016.

[106] J. Böhm, S. Scherzer, E. Krol, I. Kreuzer, K. von Meyer, C. Lorey, T. D. Mueller, L. Shabala, I. Monte, R. Solano *et al.*, "The venus flytrap dionaea muscipula counts prey-induced action potentials to induce sodium uptake," *Current Biology*, vol. 26, no. 3, pp. 286–295, 2016.

[107] M. Escalante-Pérez, E. Krol, A. Stange, D. Geiger, K. A. Al-Rasheid, B. Hause, E. Neher, and R. Hedrich, "A special pair of phytohormones controls excitability, slow closure, and external stomach formation in the venus flytrap," *Proceedings of the National Academy of Sciences*, vol. 108, no. 37, pp. 15 492–15 497, 2011.

[108] P. Jamieson, K. B. Kent, F. Gharibian, and L. Shannon, "Odin ii-an open-source Verilog HDL synthesis tool for CAD research," in *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*. IEEE, 2010, pp. 149–156.

[109] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Computer Aided Verification*. Springer Berlin Heidelberg, 2010, pp. 24–40.

[110] A. Vachoux, C. Grimm, and K. Einwich, "Analog and mixed signal modelling with SystemC-AMS," in *Circuits and Systems, 2003. ISCAS'03. Proceedings of the 2003 International Symposium on*, vol. 3. IEEE, 2003, pp. III–III.

[111] M. Hucka, A. Finney, H. M. Sauro, H. Bolouri, J. C. Doyle, H. Kitano, A. P. Arkin, B. J. Bornstein, D. Bray, A. Cornish-Bowden *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.

[112] C. Lou, X. Liu, M. Ni, Y. Huang, Q. Huang, L. Huang, L. Jiang, D. Lu, M. Wang, C. Liu *et al.*, "Synthesizing a novel genetic sequential logic circuit: a push-on push-off switch," *Molecular Systems Biology*, vol. 6, no. 1, p. 350, 2010.

[113] T. S. Bayer and C. D. Smolke, "Programmable ligand-controlled riboregulators of eukaryotic gene expression," *Nature Biotechnology*, vol. 23, no. 3, p. 337, 2005.

[114] J. E. Dueber, B. J. Yeh, K. Chak, and W. A. Lim, "Reprogramming control of an allosteric signaling switch through modular recombination," *Science*, vol. 301, no. 5641, pp. 1904–1908, 2003.

[115] D. R. Burrill and P. A. Silver, "Synthetic circuit identifies subpopulations with sustained memory of DNA damage," *Genes & Development*, vol. 25, no. 5, pp. 434–439, 2011.

[116] D. R. Burrill, M. C. Inniss, P. M. Boyle, and P. A. Silver, "Synthetic memory circuits for tracking human cell fate," *Genes & Development*, vol. 26, no. 13, pp. 1486–1497, 2012.

[117] A. Davis and S. Nowick, *Encyclopedia of Computer Science and Technology*, supplement 23 ed., J. G. W. Allen Kent, Ed. CRC Press, 1998, vol. 38, no. 231–286.

[118] H. Kaeslin, *Top-Down Digital VLSI Design: From Architectures to Gate-Level Circuits and FPGAs*. Morgan Kaufmann, 2014.

[119] H. B. Bakoglu, "Circuits, interconnections, and packaging for VLSI." 1990.

[120] K. Nagaraj, A. S. Kamath, K. Subburaj, B. Chattopadhyay, G. Nayak, S. S. Evani, N. P. Nayak, I. Prathapan, F. Zhang, and B. Haroun, "Architectures and circuit techniques for multi-purpose digital phase lock loops," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 60, no. 3, pp. 517–528, 2013.

[121] B. Razavi, *DelayLocked Loops An Overview*. IEEE, 2003, vol. 1, ch. 2, pp. 13–22.

[122] C. J. Myers, W. Belluomini, K. Kallpack, E. Peskin, and H. Zheng, "Timed circuits: a new paradigm for high-speed design," in *Proceedings of the 2001 Asia and South Pacific Design Automation Conference*. ACM, 2001, pp. 335–340.

[123] J. L. Peterson, "Petri net theory and the modeling of systems," 1981.

[124] J. Kim, K. S. White, and E. Winfree, "Construction of an in vitro bistable circuit from synthetic transcriptional switches," *Molecular Systems Biology*, vol. 2, no. 1, p. 68, 2006.

[125] A. Nagy, "Cre recombinase: the universal reagent for genome tailoring," *Genesis*, vol. 26, no. 2, pp. 99–109, 2000.

[126] M. Galdzicki, K. Clancy, E. Oberortner, M. Pocock, J. Quinn, C. Rodriguez, N. Roehner, M. Wilson, L. Adam, C. Anderson *et al.*, "The synthetic biology open language (SBOL) provides a community standard for communicating designs in synthetic biology," *Nature Biotechnology*, vol. 32, no. 6, p. 545, 2014.

[127] N. Roehner, J. Beal, K. Clancy, B. Bartley, G. Mısırlı, R. Grünberg, E. Oberortner, M. Pocock, M. Bissell, C. Madsen, T. Nguyen, M. Zhang, Z. Zhang, Z. Zundel, D. Densmore, J. Gennari, A. Wipat, H. Sauro, and C. Myers, "Sharing structure and function in biological design with SBOL 2.0," *ACS Synthetic Biology*, vol. 5, no. 6, pp. 498–506, 2016.

[128] M. Hucka, F. T. Bergmann, S. Hoops, S. M. Keating, S. Sahle, J. C. Schaff, L. P. Smith, and D. J. Wilkinson, "The systems biology markup language (SBML): language specification for level 3 version 1 core," *Journal of Integrative Bioinformatics*, vol. 12, no. 2, pp. 382–549, 2015.

[129] R. Gauges, U. Rost, S. Sahle, K. Wengler, and F. T. Bergmann, "The systems biology markup language (SBML) level 3 package: layout, version 1 core," *Journal of Integrative Bioinformatics*, vol. 12, no. 2, pp. 550–602, 2015.

[130] L. P. Smith, M. Hucka, S. Hoops, A. Finney, M. Ginkel, C. J. Myers, I. Moraru, and W. Liebermeister, "SBML level 3 package: hierarchical model composition, version 1 release 3," *Journal of Integrative Bioinformatics*, vol. 12, no. 2, pp. 603–659, 2015.

[131] B. G. Olivier and F. T. Bergmann, "The systems biology markup language (SBML) level 3 package: flux balance constraints," *Journal of Integrative Bioinformatics*, vol. 12, no. 2, pp. 660–690, 2015.

[132] C. Chaouiya, S. M. Keating, D. Berenguier, A. Naldi, D. Thieffry, M. P. van Iersel, N. L. Novère, and T. Helikar, "The systems biology markup language (SBML) level 3 package: qualitative models, version 1, release 1," *Journal of Integrative Bioinformatics*, vol. 12, no. 2, p. 270, 2015.

[133] C. Madsen, C. J. Myers, T. Patterson, N. Roehner, J. T. Stevens, and C. Winstead, "Design and test of genetic circuits using iBioSim," *IEEE Design & Test of Computers*, vol. 29, no. 3, pp. 32–39, 2012.

[134] G. Misirli, A. Wipat, J. Mullen, K. James, M. Pocock, W. Smith, N. Allenby, and J. S. Hallinan, "BacillOndex: an integrated data resource for systems and synthetic biology," *Journal of Integrative Bioinformatics*, vol. 10, no. 2, pp. 103–116, 2013.

[135] G. Mısırlı, J. Hallinan, M. Pocock, P. Lord, J. A. McLaughlin, H. Sauro, and A. Wipat, "Data integration and mining for synthetic biology design," *ACS Synthetic Biology*, vol. 5, no. 10, pp. 1086–1097, 2016.

[136] N. Roehner, Z. Zhang, T. Nguyen, and C. J. Myers, "Generating systems biology markup language models from the synthetic biology open language," *ACS Synthetic Biology*, vol. 4, no. 8, pp. 873–879, 2015.

[137] T. Nguyen, N. Roehner, Z. Zundel, and C. J. Myers, "A converter from the systems biology markup language to the synthetic biology open language," *ACS Synthetic Biology*, vol. 5, no. 6, pp. 479–486, 2016.

[138] D. Waltemath, R. Adams, F. T. Bergmann, M. Hucka, F. Kolpakov, A. K. Miller, I. I. Moraru, D. Nickerson, S. Sahle, J. L. Snoep *et al.*, "Reproducible computational biology experiments with SED-ML-the simulation experiment description markup language," *BMC Systems Biology*, vol. 5, no. 1, p. 198, 2011.

[139] F. T. Bergmann, R. Adams, S. Moodie, J. Cooper, M. Glont, M. Golebiewski, M. Hucka, C. Laibe, A. K. Miller, D. P. Nickerson *et al.*, "COMBINE archive and OMEX format: one file to share all information to reproduce a modeling project," *BMC Bioinformatics*, vol. 15, no. 1, p. 369, 2014.

[140] N. Rodriguez, A. Thomas, L. Watanabe, I. Y. Vazirabad, V. Kofia, H. F. Gómez, F. Mittag, J. Matthes, J. Rudolph, F. Wrzodek *et al.*, "JSBML 1.0: providing a smorgasbord of options to encode systems biology models," *Bioinformatics*, vol. 31, no. 20, pp. 3383–3386, 2015.

[141] B. J. Bornstein, S. M. Keating, A. Jouraku, and M. Hucka, "LibSBML: an API library for SBML," *Bioinformatics*, vol. 24, no. 6, pp. 880–881, 2008.

[142] M. Courtot, N. Juty, C. KnÕìpfer, D. Waltemath, A. Zhukova, A. DrÕạger, M. Dumontier, A. Finney, M. Golebiewski, J. Hastings, S. Hoops, S. Keating, D. B. Kell, S. Kerrien, J. Lawson, A. Lister, J. Lu, R. Machne, P. Mendes, M. Pocock, N. Rodriguez, A. Villeger, D. J. Wilkinson, S. Wimalaratne, C. Laibe, M. Hucka, and N. L. Novère, "Controlled vocabularies and semantics in systems biology," *Molecular Systems Biology*, vol. 7, no. 1, p. 543, jan 2011.

[143] C. J. Myers, *Asynchronous Circuit Design*. John Wiley & Sons, 2001.

[144] E. V. Nikolaev and E. D. Sontag, "Quorum-sensing synchronization of synthetic toggle switches: a design based on monotone dynamical systems theory," *PLoS Computational Biology*, vol. 12, no. 4, p. e1004881, 2016.

[145] P. Vaidyanathan, B. S. Der, S. Bhatia, N. Roehner, R. Silva, C. A. Voigt, and D. Densmore, "A framework for genetic logic synthesis," *Proceedings of the IEEE*, vol. 103, no. 11, pp. 2196–2207, 2015.

[146] A. Tamsir, J. J. Tabor, and C. A. Voigt, "Robust multicellular computing using genetically encoded NOR gates and chemical 'wires'," *Nature*, vol. 469, no. 7329, p. 212, 2011.

[147] J. A. McLaughlin, C. J. Myers, Z. Zundel, G. Mısırlı, M. Zhang, I. D. Ofiteru, A. Goñi-Moreno, and A. Wipat, "SynBioHub: a standards-enabled design repository for synthetic biology," *ACS Synthetic Biology*, vol. 7, no. 2, pp. 682–688, jan 2018.

[148] N. Roehner, B. Bartley, J. Beal, J. McLaughlin, M. Pocock, M. Zhang, Z. Zundel, and C. J. Myers, "Specifying combinatorial designs with the synthetic biology open language (SBOL)," *ACS Synthetic Biology*, vol. 8, no. 7, pp. 1519–1523, jun 2019.

[149] G. Misirli, T. Nguyen, J. A. McLaughlin, P. Vaidyanathan, T. S. Jones, D. Densmore, C. Myers, and A. Wipat, "A computational workflow for the automated generation of models of genetic designs," *ACS Synthetic Biology*, vol. 8, no. 7, pp. 1548–1559, 2018.

[150] C. J. Myers, *Engineering Genetic Circuits*. Chapman and Hall/CRC, 2016.

[151] ——, *Platforms for genetic design automation*. Elsevier, 2013, vol. 40, ch. 7, pp. 177–202.

[152] N.-p. Nguyen, C. Myers, H. Kuwahara, C. Winstead, and J. Keener, "Design and analysis of a robust genetic Muller C-element," *Journal of Theoretical Biology*, vol. 264, no. 2, pp. 174–187, 2010.

[153] M. Uhlén, L. Fagerberg, B. M. Hallström, C. Lindskog, P. Oksvold, A. Mardinoglu, Å. Sivertsson, C. Kampf, E. Sjöstedt, A. Asplund *et al.*, "Tissue-based map of the human proteome," *Science*, vol. 347, no. 6220, p. 1260419, 2015.

[154] M. Uhlen, P. Oksvold, L. Fagerberg, E. Lundberg, K. Jonasson, M. Forsberg, M. Zwahlen, C. Kampf, K. Wester, S. Hober *et al.*, "Towards a knowledge-based human protein atlas," *Nature Biotechnology*, vol. 28, no. 12, p. 1248, 2010.

[155] J. Zhu, G. Chen, S. Zhu, S. Li, Z. Wen, B. Li, Y. Zheng, and L. Shi, "Identification of tissue-specific protein-coding and noncoding transcripts across 14 human tissues using RNA-seq," *Scientific Reports*, vol. 6, p. 28400, 2016.

[156] I. Espinoza, M. J. Sakiyama, T. Ma, L. Fair, X. Zhou, M. Hassan, J. Zabaleta, and C. R. Gomez, "Hypoxia on the expression of hepatoma upregulated protein in prostate cancer cells," *Frontiers in Oncology*, vol. 6, p. 144, 2016.

[157] M. Duffy, A. van Dalen, C. Haglund, L. Hansson, R. Klapdor, R. Lamerz, O. Nilsson, C. Sturgeon, and O. Topolcan, "Clinical utility of biochemical markers in colorectal cancer: European group on tumour markers (EGTM) guidelines," *European Journal of Cancer*, vol. 39, no. 6, pp. 718–727, 2003.

[158] K. Goonetilleke and A. Siriwardena, "Systematic review of carbohydrate antigen (CA 19-9) as a biochemical marker in the diagnosis of pancreatic cancer," *European Journal of Surgical Oncology (EJSO)*, vol. 33, no. 3, pp. 266–270, 2007.

[159] J. Davison, "Genetic exchange between bacteria in the environment," *Plasmid*, vol. 42, no. 2, pp. 73–91, 1999.

[160] A. Pai, Y. Tanouchi, C. H. Collins, and L. You, "Engineering multicellular systems by cell–cell communication," *Current Opinion in Biotechnology*, vol. 20, no. 4, pp. 461–470, 2009.

[161] K. Brenner, L. You, and F. H. Arnold, "Engineering microbial consortia: a new frontier in synthetic biology," *Trends in Biotechnology*, vol. 26, no. 9, pp. 483–489, 2008.

[162] E. Lehman, Y. Watanabe, J. Grodstein, and H. Harkness, "Logic decomposition during technology mapping," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, no. 8, pp. 813–834, 1997.