Recombinase-Based Circuit Design Using Grammar and Binary Trees

Michael Quintin
1 May 2015

Abstract

The design of biological circuits is a rapidly growing area of study in the field of synthetic biology. By introducing multiple interacting genes into an organism, complex behaviors can be elicited. For example, a bacterium may be turned into a "biosensor" that expresses a fluorescent reporter in the presence of some stimulus.

Synthetically created gene interactions can be modeled as Boolean logic functions in much the same way as one would visualize designing an electrical circuit. Here we describe a methodology to programmatically create high-level complex gene circuit designs in which logic gates are controlled by genomic rearrangement by recombinases. This work will allow biologists to quickly create optimized gene circuits in cells to produce environmental context-dependent behavior.

Background

The term "recombinase" is used here to refer to a class of enzymes which perform conservative site specific recombination. Individually these may be integrases or excisionases which typically form complexes to rearrange DNA molecules at specific attachment sites. A recombinase will act on a pair of loci, and the result of the action is dependent on the orientation of these loci. If they are oriented in the same direction on the genome, the region between them will be excised. If they are oriented in opposing directions, the intermediate region will become inverted. Recombinases can also integrate a circular DNA molecule expressing one attachment site at the location of the corresponding site in the genome, but this particular behavior will not be used in the circuits described here. [1]

Previous work has shown how all two-input-one-output Boolean logic gates can be genetically encoded by using two pairs of recombination sites along with constitutive promoters and terminators [3]. Construction of these gates in sequence or their use to control expression of transcription factors allows for elaborate truth tables to be expressed by reprogrammed cells. Because the effect of recombinases is dependent on the relative orientation of target sites, sequence inversion can also be used to dictate states of the system with dependence on the order of the recombinase activating stimuli [3].

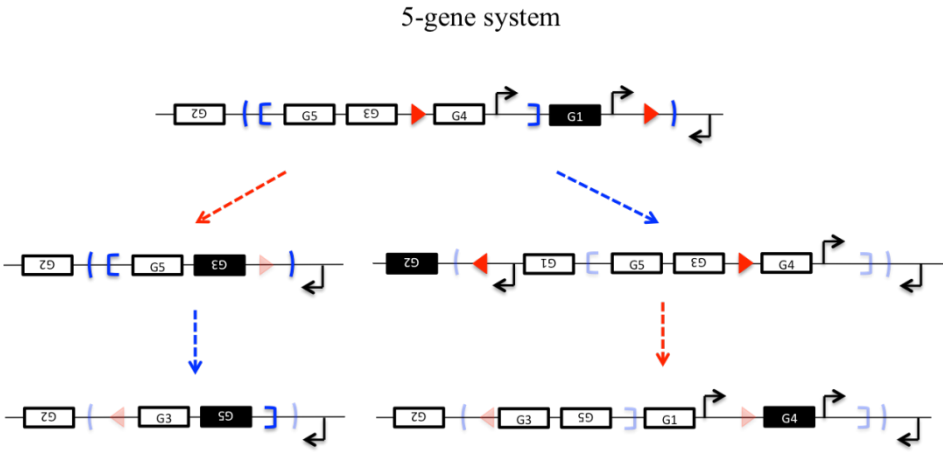

Figure 1: A genetic circuit expressing genes differentially in each of five states, achieved by order-dependent activation of recombinases. Image courtesy of Timothy Lu [unpublished results].

For simple circuit designs, it is not difficult to iterate all possible configurations of parts and calculate their outputs, or to assemble non optimized circuits that are "good enough." However, as the number of discrete states an experimenter wishes to allow increases, the set of possible configurations grows as an exponential function of the number of parts required. It is necessary to devise a way to traverse this problem space that will allow discovery of optimal arrangements for an arbitrary arrangement of states. In this paper we describe a context-free grammar (CFG) to be used as a programmatic basis for generating recombinase circuits. A context-free grammar is a set of production rules for strings of characters which fit some specific criteria. A language is defined as the set of all strings produced by a CFG.

Formally, a CFG is defined as the tuple $G = (V, \Sigma, R, S)$. V is a finite set of nonterminal characters. Σ is the alphabet of the language- that is, the set of terminal characters. R is a set of productions of the form $A \rightarrow a$ that describe transformations that replace the nonterminal character A with a series of characters, either terminals, nonterminals, or the empty string ϵ. The parameter $S \in V$ is simply the start state.

Methodology

Designing the Grammar

We designed a program in which a gene circuit is described as a string as constructed by our chosen context-free grammar. The alphabet Σ in this language corresponds to each part type: *g* for a gene, *t* for a uniderectional terminator, *r* for a recombination site. Because the interaction of each part depends on its orientation, unmarked characters will be interpreted as having forward orientation on the primary strand, and characters followed by an apostrophe will be read as having the reversed orientation placing them on the complimentary strand. Thus the sequence *pt'g* would be read as "promoter, inverted terminator, gene." Because the terminator is inverted, the gene will be expressed.

Table 1 shows the productions used in the standard form of our grammar, using the nonterminals $V = \{S, R\}$. The essential feature here that recombination sites are always created in pairs, and always correspond to exactly one partner. In practice, our program

generates an ID for each pair of loci at the time they are generated, and tracks these IDs independently of the CFG parser. This behavior obviously violates the literal interpretation of "context-free," but it is justified as a shortcut to improve performance and ease of interpretation. Either of two alternatives could work to replicate the information stored by this cheat: 1) the parser which calculates the truth table for a circuit can iterate over every possible pairing, or 2) dynamically expand the grammar to include a new pair of terminal symbols for each pair.

Standard Form				**CNF**			
$S \rightarrow$	gS	g'S		$S \rightarrow$	GS	$Y^* \rightarrow$	XR
	pS	p'S			PS	$X \rightarrow$	RS
	tS	t'S			TS	$G \rightarrow$	g
	RSRS	ϵ			XX		g'
$R \rightarrow$	r	r'			ϵ	$P \rightarrow$	p
	RSRSR			$R \rightarrow$	r		p'
					r'	$T \rightarrow$	t
					RY*		t'

Table 1: The productions set R for different versions of the grammar. These grammars are equivalent; that is, they produce the same language. Lower case terminals are: g=protein-coding sequence (gene), p=constitutive promoter, t=unidirectional terminator. Apostrophes denote an inverted element. Colors are used to track associated pairs of recombination sites.

The CYK Algorithm

A common task pertaining to CFGs is to determine whether a given word satisfies the constraints of its language. One well-known method for making this determination is the CYK algorithm (for Cocke, Younger, and Kasumi) [1]. This algorithm can also be modified to address related problems: generating the parsing tree that creates the word in question, or creating a parse tree that serves as a generator for every legal word in the language. This final problem is the version of the algorithm that we are interested in- by parsing the tree we can derive every gene circuit in an ordered manner.

Though it is useful for our purposes, the CYK algorithm has a prerequisite: the CFG must be arranged in Chomsky Normal Form (CNF). In order to convert a grammar to CNF, the productions must be of specific forms: A nonterminal can convert to either a pair of nonterminals, or a single terminal character. In addition no nonterminals except for the starting character may be nullable.

The algorithm for the CYK algorithm is as follows [2]: given a CFG $G = (V, \Sigma, R, S)$ and a word $w = a_1 \dots a_n$

Initialize the $n * n$ matrix T with all elements set to false

$$
\begin{aligned}
&\text{for } i = 1 \dots n \\
&\quad T_{i,i} = \{A \in V | A \to a_i\} \quad \textit{the set of nonterminals which can produce the letter } a_i \\
&\text{for } j = 2 \dots n \\
&\quad \text{for } i = j - 1 \dots 1 \\
&\quad\quad T_{i,j} = \emptyset \\
&\quad\quad \text{for } h = i \dots j - 1 \\
&\quad\quad\quad \text{for all productions } A \to BC \\
&\quad\quad\quad\quad \text{if } B \in T_{i,h} \text{ and } C \in T_{h+1,j} \\
&\quad\quad\quad\quad\quad T_{i,j} = T_{i,j} \cup \{A\} \quad \textit{the nonterminals which produce } BC \\
&\text{if } S \in T_{1,n} \\
&\quad w \in L(G)
\end{aligned}
$$

This algorithm has features which will appear familiar to those with experience solving hidden Markov models. The table T is populated by each cell looking only at its "descendants" to the right and bottom independent of the rest of the table. Importantly, the Markov property holds that if B and C can be derived from A, then any word derived from B followed by any word derived from C can also be derived from A. This seems intuitive, but it is essential for how we will parse the generator tree.

Earlier we asserted that the CYK algorithm can be used to create a generator for all words in a language. Because the language allows for words of any length, we will periodically pause the generation process to see if we have built a word that satisfies our target criteria. This is done with an inversion of the CYK process. Whereas the nonterminals in CYK are determined by looking at the descendant nodes and filling in the characters that can produce them, in the generator the nonterminals are decided first.

Instead of placing nonterminals in the matrix as we build the parse tree from the leaves to the root, we will instead be constructing the full tree starting at the root. Also considering that the generator can make infinitely large words, we want to make sure we check the output periodically and terminate when a solution has been found. The technique we implemented goes as follows:

1. Initialize a single root node . Each node will contain two values: v will express nonterminals, and w will hold this node's sublanguage: the set of words that can be generated from a node and all its children. Set $v_0 = S$.

2. For every leaf node (having no children) N where $|v_N| = 1$, set $w_N = \{a \in \Sigma^* | v_N \to a\}$ the set of terminal characters that can be produced in a single step by the nonterminal.

3. For every branch node (having children) B at a depth 1 away from the maximum:
   a. If $|v_B| = 1$, set $w_B = w_B \cup \{w_C | C \in children(B)\}$
   b. If $|v_B| = 2$, set $w_B = w_B \cup \{w_C w_D | C, D \in children(B) \cap v_{B1} \to v_C \cap v_{B2} \to v_D\}$
   c. Recursively update this node's ancestors in the same way until w_S has been updated.
   d. Parse each new member of w_S to see if a satisfactory word has been generated. If so, terminate the process.

4. For each leaf node, for each production $v_{Ni} \rightarrow a \in V$, create a new child node with $v = a$.

5. Repeat from step 2 until an answer is found.

The final part of the procedure is to query for functional gene circuits. The input criteria are a series of two-input Boolean statements of the form e.g. "if RecA AND RecB then GeneA." The system does not impose any limit on the size or number of statements.

In part 3d of the procedure above, we examine a set of strings to see if any match the input statements. A truth table for each given gene's activation is created by scanning for promoters followed by genes with the same orientation without terminators between them. Then each pair of recombinases is "activated," shuffling the string by inversion or excision, until every configuration has been scanned.

Analysis

Data Properties

The tree parsing algorithm above describes a breadth-first search for all gene circuits that can be made with the four basic parts used. The essential property of the hierarchy that the tree is built on is that each child node typically adds only one symbol to the output string, with the one exception being a two-terminal node that never has any children. This sets an upper bound for the length of a word generated by the inclusion of a given node: the number of parts is at most equal to the depth of the node. By only increasing the search depth after all other options have been exhausted, we take the most conservative approach in terms of word length.

Another helpful emergent property of the slow emergence of new parts is that recombination sites are the most "complicated" to add, requiring three productions to get from $S \rightarrow r$ while all other symbols only require two. The generator will minimize the number of recombination sites if an equivalent circuit can be built with a similar number of alternate parts. This is useful because the library of available recombinases is limited.

Runtime Analysis

Parsing a word to derive its truth table requires that it be read once in each possible configuration. The possibility for recombinase pairs to overlap introduces an order dependence on the system, so the number of states in the worst-case scenario is $\left(n\ recombination\ site\ s/2\right)^2$.

Updating the tree to add a node, and updating the words produced in a single node's sublanguage v are linear-time processes. A node can have at most four children (there are four productions from S), so defining the start node as depth 0 the upper limit of the number of nodes at depth $d = 4^d$. Cascading the change in sublanguages upwards happens once to each node for every node added, making its execution time $O(n^2)$.

Suiti et al. showed that all two-input logic gates can be created with at most six additions of parts- the largest gate is the NOR gate, requiring two terminators, two pairs of recombination sites, and a single promoter and gene [3]. The NOR gate requires nine

productions to be produced in our grammar. We can calculate the maximum number of gates needed in the worst case to be the number of atomic Boolean statements b in the input. This gives us an approximation of the maximum depth of the tree, $9 * b$.

Combining the above, we expect the absolute worst-case runtime of the algorithm presented here to necessitate $\left(\frac{b}{2}\right)^2$ string parses for each of 4^{9b} words, and an additional $\left(4^{9b}\right)^2$ update steps; our algorithm has a worst-case runtime on the order of $O(4^{9b})$. This is a massive number, though in practice we expect that the gates will typically be simpler than the NOR gate and that there will be an equivalent and much simpler Boolean statement expressing the desired truth table, which must be found by our exhaustive search.

Future Work

The immediate application of this work will be its integration into a web-based interface that will allow researchers to quickly define a set of conditions and generate the corresponding genetic circuit. Further development will focus first on improving the algorithm's runtime. Future goals will include examining alternate grammar structures that may have other useful properties such as the part number minimization seen here, and devising alternate criteria to score the produced circuits.

Citations

1. Alberts, B. et al. 2007. "Molecular Biology of the Cell." 5th ed. Garland Science. pp 322-325
2. Lange, M. and Leiß, H. 2009. "To CNF or not to CNF? An Efficient Yet Presentable Version of the CYK Algorithm." *Informatica Didactica* 8.
3. Siuti, P., Yazbek, J., and Lu, T.K. 2013. "Synthetic circuits integrating logic and memory in living cells." *Nature Biotechnology* 31:5. pp 448-452