Computing Threshold Circuits with Void Reactions in Step Chemical Reaction Networks*

Rachel Anderson[1], Alberto Avila[1], Bin Fu[1], Timothy Gomez[2], Elise Grizzell[1], Aiden Massie[1], Gourab Mukhopadhyay[1], Adrian Salinas[1], Robert Schweller[1], Evan Tomai[3], and Tim Wylie[1]

[1]University of Texas Rio Grande Valley
[2]Massachusetts Institute of Technology
[3]University of Texas Dallas

Abstract

We introduce a new model of *step* Chemical Reaction Networks (step CRNs), motivated by the stepwise addition of materials in standard lab procedures. Step CRNs have ordered reactants that transform into products via reaction rules over a series of steps. We study an important subset of weak reaction rules, *void* rules, in which chemical species may only be deleted but never changed. We demonstrate the capabilities of these simple limited systems to simulate threshold circuits and compute functions using various configurations of rule sizes and step constructions, and prove that without steps, void rules are incapable of these computations, which further motivates the step model. Additionally, we prove the coNP-completeness of verifying if a given step CRN computes a function, holding even for $O(1)$ step systems.

1 Introduction

Chemical Reaction Networks (CRNs) are one of the most established and longest studied models of self-assembly [5, 6]. CRNs originate in attempting to model chemical interactions as molecular species that react and create products from the reaction. This can be represented as an original number of each species and a set of replacement rules. The fundamental nature of the model is evident in the independent inception of equivalent models in multiple areas of research through other motivations [17], such as Vector Addition Systems (VASs) [26] and Petri-Nets [32]. Further, Population Protocols [2] are a restricted version where the number of input and output elements are each two.

Step CRNs. We propose and investigate an important but straightforward extension to the CRN model motivated by the desire to reflect standard laboratory and medical practices. The *Step* CRN models augments the CRN model with a sequence of discrete steps where an additional specified amount of chemical species is combined with the existing CRN after running the system to completion. Our goal is to explore the computational power of Step CRNs using highly restricted classes of CRN rules that would otherwise be computationally weak. In particular, we consider the problem of implementing the computationally universal class of *Threshold Circuits* using only *void* rules.

Void Rules. We study the computational power of Step CRNs under an extremely simple subset of CRN rules termed *void* rules [1]. General CRN rules are powerful since they allow the removal, addition, and replacement of species. Impressively, these rules have successful experimental implementations using DNA strand replacement mechanisms [37]. However, implementing this level of generality requires sophisticated, and large, DNA complexes that incur practical errors and constitute one of the primary hurdles limiting

*This research was supported in part by National Science Foundation Grant CCF-2329918.

the scalable implementation of molecular computing schemes [13, 43]. In contrast, void rules only have the ability to delete species. Such a simple subset of reactions could plausibly permit drastically simpler and scalable molecular implementations based solely on the pair-wise bonding strength of single-stranded DNA. The only drawback is the inability of void rule systems to compute even simple functions. We show that void rules become computationally powerful in the step model with just tri-molecular or bi-molecular interactions. Specifically, we show how Threshold Circuits (TC), a powerful class of circuits with applications in deep learning, are simulated with void rules using a number of steps linear in the circuit's depth. Our utilization of steps under this void rule restriction is necessary, as we further show that even simple circuits require the use of steps when restricted to pure void rules.

1.1 Previous Work

Computation in Chemical Reaction Networks. Stochastic Chemical Reaction Networks are only Turing-complete with the possibility for error [36] while error-free stochastic Chemical Reaction Networks can compute precisely the set of semilinear functions [4, 15]. CRNs have also recently been shown to be experimentally viable through DNA Strand Displacement (DSD) systems [37], and several CRN to DSD compilers have been created [9, 27, 40, 46].

Boolean Circuits. Using molecules for information storage and Boolean logic is a deep field of study. Here, we show a few highlights, starting with one of the first discussions in 1988 [8] and an initial presentation of circuits with CRNs in 1991 [24]. Since then, the area has been extensively studied in CRNs and related models [7, 10, 12, 17, 20, 25, 28, 33, 34]. Numerous gates have been built experimentally and proposed theoretically such as the AND [12, 18, 29, 34, 39, 45], OR [12, 18, 34, 39], NOT [12], XOR [12, 45], NAND [12, 17, 20, 44], NOR [12], Parity [21, 22, 23], and Majority [3, 11, 30]. Symmetric boolean functions of n variables such as Majority have been found to have a circuit depth of $O(\log n)$ when implemented by AND, OR and NOT gates [35].

Void Rules. The reachability problem, with systems of only void rules in proper CRNs, was studied in [1]. Previous studies had included void rules as a part of their systems but were never studied exclusively. The CRN++ programming language [42] allows these reactions to be programmed using the *sub* module. However, if any product(s) remain, they can differ from the reactants. They can also be considered a subcategory of the broader concept of the extinction of rules and species in a system as referred to in [44].

Mixing Systems. Another generalization of CRNs that is closely related to the step model is I/O CRNs [20], where additional inputs can be added at timed intervals. Still, those inputs are read-only in the system (used exclusively as catalysts). Step CRNs generalize I/O CRNs as the inputs are not read-only and are rate-independent, unlike I/O CRNs. Staged systems have been explored in many self-assembly models[14, 16, 19, 31].

1.2 Our Contributions

Table 1 has an overview of the main results of this paper beyond the introduction of the model and simulation definitions. The most important results being the ability to simulate the class of Threshold Circuits (TC) by simulating AND, OR, NOT, and MAJORITY gates through a restrictive definition of simulation while only using small void rules.

In Section 2, we define Step Chemical Reaction Networks and what it means to compute a function. Following, in Section 3, we show how to simulate the class TC of Threshold Circuits with void rules of size $(3, 0)$ (rules where 3 species react to delete each other) using $O(D \log f)$ steps, where D is the depth of the circuit and f denotes the maximum fan-out of the circuit. In Section 4, we achieve the same result using both $(2, 0)$ and $(2, 1)$ rules and a slightly more efficient step complexity of $O(D)$. We then use exclusively $(2, 1)$ rules to achieve this same result by adding a factor of $\log F_{maj}$ to the steps, where F_{maj} is the maximum fan-in of majority gates. In Section 5, we show there exist functions that require a logarithmic number of steps when restricted to constant reaction size, as well as the existence of $O(1)$-depth threshold circuits of fan-out f that require $\Omega(\log f)$ steps, which matches the $O(D \log f)$ upper bound for $(3, 0)$

Function Computation					
Rules	**Species**	**Steps**	**Simulation**	**Family**	**Ref**
$(3,0)$	$O(\min(W^2, GF_{out}))$	$O(D \log F_{out})$	Strict	**TC** Circuits	Theorem 3.3
$(2,0)(2,1)$	$O(G)$	$O(D)$	Strict	**TC** Circuits	Theorem 4.1
$(2,1)$	$O(G)$	$O(D \log F_{maj})$	Strict	**TC** Circuits	Corollary 4.2
$(c,0)$	any	$\Omega(\log k)$	Strict	k-CNOT	Theorem 5.2

Strict Function Verification			
Rules	**Steps**	**Complexity**	**Ref**
$(3,0)$	$O(1)$	coNP-complete	Theorem 5.5

Table 1: Summary of n-bit circuit simulation results. D is the depth of the circuit, W is the width, G is the number of gates in a circuit or number of operators in a formula, F_{out} is the max fan-out, and F_{maj} is the max fan-in of majority gates. **TC** stands for Threshold Circuits. The k-CNOT is a k fan-in generalization of a Controlled NOT gate. Rule size $(c,0)$ means the row holds for all integer constants $c > 0$.
circuits. Finally, we show that it is coNP-complete to know whether a function can be strictly simulated by a step CRN system.

2 Preliminaries

2.1 Chemical Reaction Networks

Basics. Let $\Lambda = \{\lambda_1, \lambda_2, \ldots, \lambda_{|\Lambda|}\}$ denote some ordered alphabet of *species*. A configuration over Λ is a length-$|\Lambda|$ vector of non-negative integers that denotes the number of copies of each present species. A *rule* or *reaction* is represented as an ordered pair of configuration vectors $R = (R_r, R_p)$. R_r contains the minimum counts of each *reactant* species necessary for reaction R to occur, where reactant species are either *consumed* by the rule in some count or leveraged as *catalysts* (not consumed); in some cases a combination of the two. The *product* vector R_p has the count of each species *produced* by the *application* of rule R, effectively replacing vector R_r. The species corresponding to the non-zero elements of R_r and R_p are termed *reactants* and *products* of R, respectively.

The *application* vector of R is $R_a = R_p - R_r$, which shows the net change in species counts after applying rule R once. For a configuration C and rule R, we say R is applicable to C if $C[i] \geq R_r[i]$ for all $1 \leq i \leq |\Lambda|$, and we define the *application* of R to C as the configuration $C' = C + R_a$. For a set of rules Γ, a configuration C, and rule $R \in \Gamma$ applicable to C that produces $C' = C + R_a$, we say $C \to^1_\Gamma C'$, a relation denoting that C can transition to C' by way of a single rule application from Γ. We further use the notation $C \to^*_\Gamma C'$ to signify the transitive closure of $\to^1_\Gamma$ and say C' is *reachable* from C under Γ, i.e., C' can be reached by applying a sequence of applicable rules from Γ to initial configuration C. Here, we use the following notation to depict a rule $R = (R_r, R_p)$: $\sum_{i=1}^{|\Lambda|} R_r[i]s_i \to \sum_{i=1}^{|\Lambda|} R_p[i]s_i$.

For example, a rule turning two copies of species H and one copy of species O into one copy of species W would be written as $2H + O \to W$.

Definition 2.1 (Discrete Chemical Reaction Network)**.** *A discrete chemical reaction network (CRN) is an ordered pair* (Λ, Γ) *where* Λ *is an ordered alphabet of species, and* Γ *is a set of rules over* Λ*.*

An initial configuration and CRN (Λ, Γ) is said to be *bounded* if a terminal configuration is guaranteed to be reached within some finite number of rule applications starting from configuration I. We denote the set of reachable configurations of a CRN as $REACH_{I,\Lambda,\Gamma}$. A configuration is called *terminal* with respect to a CRN (Λ, Γ) if no rule R can be applied to it. We define the subset of reachable configurations that are terminal as $TERM_{I,\Lambda,\Gamma}$.

2.2 Void Rules

Definition 2.2 (Void and Autogenesis rules)**.** *A rule* $R = (R_r, R_p)$ *is a* void *rule if* $R_a = R_p - R_r$ *has no positive entries and at least one negative entry. A rule is an* autogenesis *rule if* R_a *has no negative values*

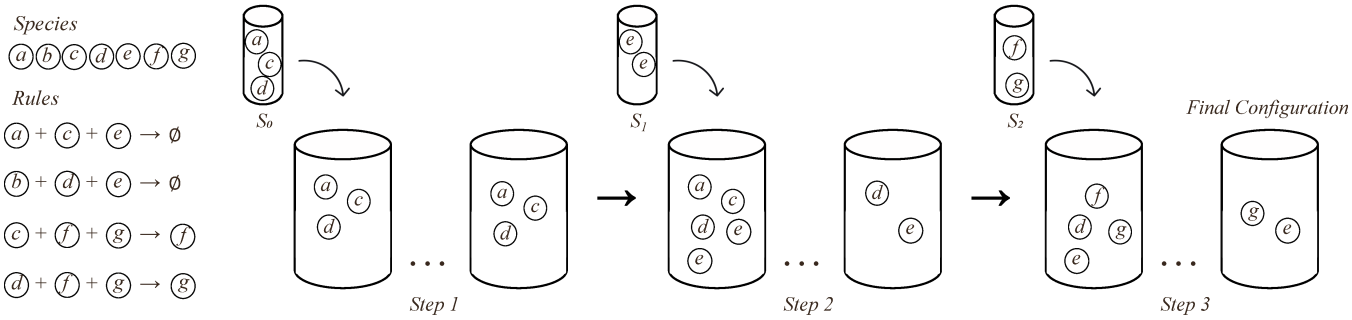


Figure 1: An example step CRN system. The test tubes show the species added at each step and the system with those elements added. The CRN species and void rule-set are shown on the left.

and at least one positive value. If the reactants and products of a rule are each multisets, a void rule is a rule whose product multiset is a strict submultiset of the reactants, and an autogenesis rule one where the reactants are a strict submultiset of the products. There are two classes of void rules, catalytic and true void. In catalytic void rules, such as $(2,1)$ *rules, one or more reactants remain, and one or more is deleted after the rule is applied. In true void rules, such as* $(2,0)$ *and* $(3,0)$ *rules, there are no products remaining.*

Definition 2.3. *The* size/volume *of a configuration vector* C *is* $\texttt{volume}(C) = \sum C[i]$.

Definition 2.4 (size-(i,j) rules)**.** *A rule* $R = (R_r, R_p)$ *is said to be a size-*(i,j) *rule if* $(i,j) = (\texttt{volume}(R_r), \texttt{volume}(R_p))$. *A reaction is trimolecular if* $i = 3$*, bimolecular if* $i = 2$*, and unimolecular if* $i = 1$.

2.3 Step CRNs

A step CRN is an augmentation of a basic CRN in which a sequence of additional copies of some system species are added after a terminal configuration is reached. Formally, a step CRN of k steps is an ordered pair $((\Lambda, \Gamma), (s_0, s_1, s_2, \ldots, s_{k-1}))$, where the first element of the pair is a normal CRN (Λ, Γ), and the second is a sequence of length-$|\Lambda|$ vectors of non-negative integers denoting how many copies of each species type to add after each step. Figure 1 illustrates a simple step CRN system.

Given a step CRN, we define the set of reachable configurations after each sequential step. Let $REACH_1$ be the set of reachable configurations of (Λ, Γ) with initial configuration c_0 at step s_0, and let $TERM_1$ be the subset of reachable configurations that are terminal. Define $REACH_2$ to be the union of all reachable configurations from each possible starting configuration attained by adding s_1 to a configuration in $TERM_1$. Let $TERM_2$ be the subset of these reachable configurations that are terminal. Similarly, define $REACH_i$ to be the union of all reachable sets attained by using initial configuration c_{i-1} at step s_{i-1} plus any element of $TERM_{i-1}$, and let $TERM_i$ denote the subset of these configurations that are terminal.

The set of reachable configurations for a k-step CRN is the set $REACH_k$, and the set of terminal configurations is $TERM_k$. A classical CRN can be represented as a step CRN with $k = 1$ steps and an initial configuration of $I = s_0$.

Note that our definitions assume only the terminal configurations of a given step are passed on to seed the subsequent step. This makes sense if we assume we are dealing with *bounded* systems, as this represents simply waiting long enough for all configurations to reach a terminal state before proceeding to the next step. In this paper we only consider bounded void rule systems; we leave more general definitions to be discussed in future work.

2.4 Computing Functions in Step CRNs

Here, we define what it means for a step CRN to compute a function $f(x_1, \ldots_n) = (y_1, \ldots y_m)$ that maps n-bit strings to m-bit strings. For each input bit, we denote two separate species types, one representing bit 0, and the other bit 1. We add one copy for each bit to encode an input n-bit strig. Similarly, each

output bit has two representatives(for 0 and 1), and we say the step CRN computes function f if for any given n-bit input $x_1, \dots x_n$, the system results in a final configuration whose output species encode the string $f(x_1, \dots x_n)$. For a fixed function f, the values denoted at each step s_i are fixed to disallow outside computation.

Input-Strict Step CRN Computing. Given a Boolean function $f(x_1, \dots, x_n) = (y_1, \dots, y_m)$ that maps a string of n bits to a string of m bits, we define the computation of f with a step CRN. An input-strict step CRN computer is a tuple $C_s = (S, X, Y)$ where $S = ((\Lambda, \Gamma), (s_0, s_1, \dots, s_{k-1}))$ is a step CRN, and $X = ((x_1^0, x_1^1), \dots, (x_n^0, x_n^1))$ and $Y = ((y_1^0, y_1^1), \dots, (y_m^0, y_m^1))$ are sequences of ordered-pairs with each $x_i^0, x_i^1, y_j^0, y_j^1 \in \Lambda$. Given an n-input bit string $b = b_1, \dots, b_n$, configuration $X(b)$ is defined as the configuration over Λ obtained by including one copy of x_i^0 only if $b_i = 0$ and one copy of x_i^1 only if $b_i = 1$ for each bit b_i. We consider this representation of the input to be strict, as opposed to allowing multiple copies of each input bit species. The corresponding step CRN $(\Lambda, \Gamma, (s_0 + X(b), \dots, s_{k-1}))$ is obtained by adding $X(b)$ to s_0 in the first step, which conceptually represents the system programmed with specific input b.

An input-strict step CRN computer *computes* function f if, for all n-bit strings b and for all terminal configurations of $(\Lambda, \Gamma, (s_0 + X(b), \dots, s_{k-1}))$, the terminal configuration contains at least 1 copy of y_j^0 and 0 copies of y_j^1 if the j^{th} bit of $f(b)$ is 0, and at least 1 copy of y_j^1 and 0 copies of y_j^0 if the j^{th} bit of $f(b)$ is 1, for all j from 1 to m.

We use the term *strict* to denote requiring exactly one copy of each bit species and we leave it for future work to consider a more general form of input allowance or strict output. Here, we only consider input-strict computation, so we use input-strict and strict interchangeably.

Relation to CRN Computers. Previous models of CRN computers considered functions over large domains such as the positive integers. Due to the infinite domain, the input to such systems cannot be bounded. As such, the CRN computers shown in [15] define the input in terms of the volume of some input species. In these scenarios, CRN computers are limited to computing semi-linear functions. Here, we instead focus on computing n-bit functions, and instead encode the input per bit with potentially unique species. This is a model more similar to the PSPACE computer shown in [38].

2.5 Boolean and Threshold Circuits

A Boolean circuit on n variables $x_1, x_2, \dots, x_n$ is a directed, acyclic multi-graph. The vertices of the graph are generally referred to as *gates*. The in-degree and out-degree of a gate are called the *fan-in* and *fan-out* of the gate respectively. The fan-in 0 gates (*source* gates) are labeled from $x_1, x_2, \dots, x_n$, or labeled by constants starting at 0 or 1. Each non-source gate is labeled with a function name, such as AND, OR, or NOT. Given an assignment of boolean values to variables $x_1, x_2, \dots, x_n$, each gate in the circuit can be assigned a value by first assigning all source vertices the value matching the labeled constant or labeled variable value and subsequently assigning each gate the value computed by its labeled function on the values of its children. Given a fixed ordering on the output gates, the sequence of bits assigned to the output gates denotes the value computed by the circuit on the given input.

The *depth* of a circuit is the longest path from a source vertex to an output vertex. Here, we focus on circuits that consist of AND, OR, NOT, and MAJORITY gates with arbitrary fan in. We refer to circuits that use these gates as *threshold circuits* (TC).

2.6 Notation

When discussing a boolean circuit, we use the following variables to denote the properties of the circuit: Let D denote the circuit's depth, G the number of gates in the circuit, W the circuit's width, F_{out} the maximum fan-out of any gate in the circuit, F_{in} the maximum fan-in, and F_{maj} the maximum fan-in of any majority gate within the circuit.

3 Computation of Threshold Circuits with (3, 0) Rules

Here, we introduce a step CRN system construction with only true void rules that can compute Threshold Circuits. In other words, given any TC and some truth assignment to the input variables, we can construct a step CRN with only true void rules that computes the same output as the circuit.

This section specifically focuses on step CRNs consisting of $(3, 0)$ rules. Subsection 3.1 shows how the system can compute individual logic gates, and we show an example construction of a circuit in Subsection 3.2. We then present the general construction of computing TC circuits by two different methods, differing in the number of species needed based on the fan-out and width of the circuit. This results in Theorem 3.3, which states that TCs can be strictly computed, even with unbounded fan-out, with $O(\min(W^2, GF_{out}))$ species, $O(D \log F_{out})$ steps, and $O(W)$ volume.

3.1 Computing Logic Gates

Indexing. The number of steps to compute an individual depth level of a circuit varies between 2-8 steps depending on the gates and wiring of the specified circuit. To convert a circuit into a $(3, 0)$ step CRN system, we need to *index* the wires (input and output) at each level of the circuit in order to ensure the species is input to the correct gate. An example circuit with bit/wire indexing is shown in Figure 3c. At each level, we call the indices of the inputs of gates the *input indices*, and the indices of the output of each gate the *gate indices*. Note that the index numbers may need to change along the wire, or change due to fan-out/fan-in (see Figure 3c). This is accomplished by rules of the form $t_{j\to i}$ that map an input index of j to a gate index of i.

Bit Representation. The input bits of a binary gate are represented in a step CRN with $(3, 0)$ rules by the species x_n^b, where $n \in \mathbb{N}$ and $b \in \{T, F\}$. Here, n represents the bit's index (based on the ordering of all bits into the gates) and b represents its truth value. Let f_i^{in} be the set of all the indices of input bits fanning into a gate at index i (gate indices). Let f_i^{out} be the set of all indices of the output bits fanning out of a gate at index i.

The output bits of a gate are represented by the species $y_{n,g}^b$, where n is instead the output bit's index (input index of the next level) and g denotes the gate type $g \in \{B, A, O, N, M\}$ (BUFFER, AND, OR, NOT, and MAJORITY). The BUFFER (B) represents a signal wire that changes depth without passing through a gate. For example, the outputs of an AND gate, an OR gate, and a NOT gate at index n are represented by the species $y_{n,A}^b$, $y_{n,O}^b$, and $y_{n,N}^b$, respectively.

AND/OR Gate. The general process to compute an AND gate (an OR gate is similar) is given in Table 2. First, all input species are converted into a new species $a_{i,g}^b$ (step 1). The species retains truth value b as the original input, and includes the gate index i and type of the gate g. The species $b_{i,g}^b$ is then introduced (step 2), which computes the operation of gate g across all existing species. Any species that do not share the same truth value as the gate's intended output are deleted (step 3-4). The species remaining after the operation is then converted into the correct output species (step 5). The species u_i, v_i, w_i, and $t_{j\to i}$, where j is the input index and i is the gate index, are used to assist in removing excess species in certain steps.

AND Example. Consider an AND gate whose gate index is 1 with input bits 1 and 0 as shown in Figure 2. In this case, $|f_i^{in}| = 2$ and the initial configuration consists of the species x_1^T and x_2^F. Following Table 2, this gate can be computed in five steps.

1. Two $a_{1,A}^T$, two $a_{1,A}^F$, one $t_{1\to 1}$, and one $t_{2\to 1}$ species are added to the system. This converts the two input species of the gate into $a_{1,A}^T$ and $a_{1,A}^F$ (causes all species except $a_{1,A}^T$ and $a_{1,A}^F$ to be deleted).

2. One $b_{1,A}^T$, two $b_{1,A}^F$, and two u_1 species are added. All species except a single $b_{1,A}^F$ are deleted by reactions.

3. Four v_1 species are added to remove excess species. There are none, so no reactions occur.

4. Two w_1 are added to delete excess species. Now, only a $b_{1,A}^F$ species remains.

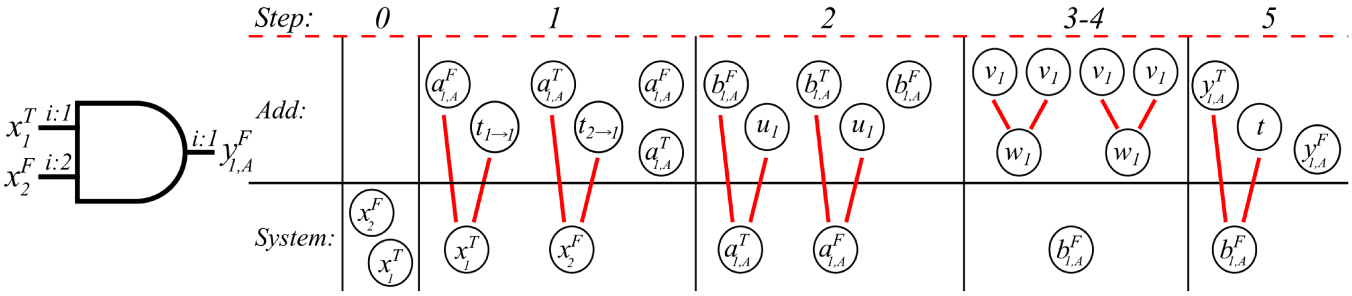


Figure 2: Example AND gate and steps to compute using $(3, 0)$ rules. Note the gate indexing of the wires ($i:1$ and $i:2$) and the input indexing for the next level ($i:1$ since there is only one gate). The process of computing the gate is shown on the right in steps. The new species added at each step are above and the remaining ones are in the system. The lines show the rules that would be executed during each step. To see the added species and rules in detail, see Table 2.

Steps			Relevant Rules	Description
1	*Add*	$\lvert f_i^{in}\rvert \cdot a_{i,g}^T$ $\lvert f_i^{in}\rvert \cdot a_{i,g}^F$ $\forall j \in f_i^{in} : t_{j\to i}$	$\forall j \in f_i^{in}$: $x_j^T + a_{i,g}^F + t_{j\to i} \to \emptyset$ $x_j^F + a_{i,g}^T + t_{j\to i} \to \emptyset$	$\forall j \in f_i^{in}$, convert x_j^b input species into $a_{i,g}^b$ species.
2	*Add*	$b_{i,g}^T$ $\lvert f_i^{in}\rvert \cdot b_{i,g}^F$ $\lvert f_i^{in}\rvert \cdot u_i$	$u_i + a_{i,g}^T + b_{i,g}^F \to \emptyset$ $u_i + a_{i,g}^F + b_{i,g}^T \to \emptyset$	Keep at least one of the correct output species and delete all incorrect species. This step computes the AND gate.*
3	*Add*	$2\lvert f_i^{in}\rvert \cdot v_i$	$u_i + v_i + v_i \to \emptyset$	Delete extra/unwanted species.
4	*Add*	$\lvert f_i^{in}\rvert \cdot w_i$	$w_i + v_i + v_i \to \emptyset$ $w_i + a_{i,g}^F + b_{i,g}^F \to \emptyset$	Delete extra/unwanted species.
5	*Add*	$y_{i,g}^T$, $y_{i,g}^F$, t	$b_{i,g}^T + y_{i,g}^F + t \to \emptyset$ $b_{i,g}^F + y_{i,g}^T + t \to \emptyset$	Convert $b_{i,g}^b$ into the proper output species $y_{i,g}^b$.

Table 2: *(3, 0) rules and steps for an AND gate. To compute an OR gate, add* $\lvert f_i^{in}\rvert \cdot b_{i,g}^T$ *and one* $b_{i,g}^F$ *in step 2 instead, and replace* $w_i + a_{i,g}^F + b_{i,g}^F \to \emptyset$ *with* $w_i + a_{i,g}^T + b_{i,g}^T \to \emptyset$ *in step 4.*

5. One $y_{1,A}^T$, one $y_{1,A}^F$ and one t species are added. The $b_{1,A}^F$ species cause the $y_{1,A}^T$ and t species to be deleted. The $y_{1,A}^F$ species is the only species remaining, which represents the intended "false" output of the AND gate.

NOT Gate. Table 3 shows the general process to computing NOT gates. To compute a NOT gate, only the output species and species t are added in. In NOT gates specifically, the input species and the output species that share the same truth value b remove each other, leaving the complement of the input species as the remaining and correct output species.

Majority Gate. The majority gate outputs 1 if and only if more than half of its inputs are 1. Otherwise, it returns 0. The general step process is overviewed in Table 4 To compute a majority gate, all input species are converted into a new species $a_{i,M}^b$ (step 1). The species retains the same index i and truth value b as the original input. If the number of species fanning into the majority gate is even, an extra *false* input species is added in. The species $b_{i,M}^b$ is then introduced, which computes the majority operation across all existing species. Any species that represent the minority inputs are deleted (step 2). The species remaining after the operation are converted into the correct output (gate index) species (step 5). The species u_i, v_i, w_i, and $t_{j\to i}$, where j is the input index and i is the gate index, are used to assist in removing excess species in certain steps.

Steps			Relevant Rules	Description
1	*Add*	$y^T_{i,N}$ $y^F_{i,N}$ $t_{j\to i}$	$y^T_{i,N} + x^T_j + t_{j\to i} \to \emptyset$ $y^F_{i,N} + x^F_j + t_{j\to i} \to \emptyset$	The output species ($y^b_{i,N}$) that is the complement of the input species (x^b_j) will be the only species remaining.

Table 3: *(3, 0) rules and steps for a NOT gate.*

Steps			Relevant Rules	Description
1	*Add*	$\lvert f^{in}_i\rvert \cdot a^T_{i,M}$ $\lvert f^{in}_i\rvert \cdot a^F_{i,M}$ $\forall j \in f^{in}_i : t_{j\to i}$	$\forall j \in f^{in}_i :$ $x^T_j + a^F_{i,M} + t_{j\to i} \to \emptyset$ $x^F_j + a^T_{i,M} + t_{j\to i} \to \emptyset$	$\forall j \in f^{in}_i$, convert x^b_j input species into $a^b_{i,M}$ species.
2	*Add*	$\lfloor \lvert f^{in}_i\rvert /2 \rfloor \cdot b^T_{i,M}$ $\lfloor \lvert f^{in}_i\rvert /2 \rfloor \cdot b^F_{i,M}$ $(\lvert f^{in}_i\rvert - 1) \cdot u_i$	$u_i + a^T_{i,M} + b^F_{i,M} \to \emptyset$ $u_i + a^F_{i,M} + b^T_{i,M} \to \emptyset$	Adding $\lfloor \lvert f^{in}_i\rvert /2 \rfloor$ amounts of $b^T_{i,M}$ and $b^F_{i,M}$ species will delete all of the minority species, leaving some amount of the majority species remaining.
3	*Add*	$2(\lvert f^{in}_i\rvert - 1) \cdot v_i$	$u_i + 2v_i \to \emptyset$	Delete extra/unwanted species.
4	*Add*	$(\lvert f^{in}_i\rvert - 1) \cdot w_i$	$w_i + 2v_i \to \emptyset$ $w_i + a^T_{i,M} + b^T_{i,M} \to \emptyset$ $w_i + a^F_{i,M} + b^F_{i,M} \to \emptyset$	Delete extra/unwanted species.
5	*Add*	$y^T_{i,M}$ $y^F_{i,M}$ t	$a^T_{i,M} + y^F_{i,M} + t \to \emptyset$ $a^F_{i,M} + y^T_{i,M} + t \to \emptyset$	Convert $a^b_{i,M}$ into the proper output species ($y^b_{i,M}$).

Table 4: *(3, 0) rules and steps for a majority gate.*

3.2 (3,0) Circuit Example

With the computation of individual gates demonstrated in our system, we now expand these features to computing entire circuits. We begin with a simple example (Figure 3c) to show the concepts before giving the general construction. The circuit has four inputs: x_1, x_2, x_3, and x_4. At the first depth layer, x_1 fans into a NOT gate and x_2 and x_3 are both fanned into an OR gate. At the next depth level, the output of the OR gate is fanned out twice. One of these outputs, along with the output of the NOT gate, is fanned into an AND gate, while the other and x_4 fans into another AND gate. At the last depth level, both AND gate outputs fan into an OR gate, which computes the final output of the circuit.

Table 5 shows how to compute the circuit in Figure 3c. The primary inputs of the circuit in Figure 3c are represented by the species in the initial configuration. Step 1 converts the primary inputs into input species. If there was any fan out of the primary inputs, it would be done in this step. Steps 2-6 compute the gates at the first depth level. Steps 7-8 compute the fan out between the first and second depth level. Step 9 converts the outputs of the gates at the first depth level into input species. Steps 10-14 use those inputs to compute the gates at the second depth level. Step 15 converts the outputs of these gates into inputs. Steps 16-20 compute the final gate. Step 21 converts the output of that gate into an input species that represents the solution to the circuit (x^F_1).

3.3 Computing Circuits

Lemma 3.1. *Threshold circuits (TC) with a max fan-out of 2 can be strictly computed by a step CRN with only (3,0) rules, $O(W^2)$ species, $O(D)$ steps, and $O(W)$ volume.*

Proof. The initial configuration of the step CRN should consist of one $y^b_{n,B}$ species for each primary input with the appropriate indices and truth values. Section 3.1 explains how to compute TC gates. In order to run a circuit, we need to convert the outputs at an index i into the inputs for the next gate at index j.

Initial Configuration:	$y^T_{1,B}$ $y^T_{2,B}$ $y^T_{3,B}$ $y^F_{4,B}$				
Steps		**Relevant Rules**	**Steps**		**Relevant Rules**
1	x^T_1, x^T_2, x^T_3, x^T_4 $t_{1\to1}$, $t_{3\to3}$ x^F_1, x^F_2, x^F_3, x^F_4 $t_{2\to2}$, $t_{4\to4}$	$y^T_{1,B} + x^F_1 + t_{1\to1} \to \emptyset$ $y^T_{2,B} + x^F_2 + t_{2\to2} \to \emptyset$ $y^T_{3,B} + x^F_3 + t_{3\to3} \to \emptyset$ $y^F_{4,B} + x^T_4 + t_{4\to4} \to \emptyset$	**10**	$2a^T_{1,A}$, $2a^T_{2,A}$ $2a^F_{1,A}$, $2a^F_{2,A}$ $t_{2\to1}$, $t_{4\to2}$ $t_{1\to1}$, $t_{3\to2}$	$x^F_1 + a^T_{1,A} + t_{1\to1} \to \emptyset$ $x^T_2 + a^F_{1,A} + t_{2\to1} \to \emptyset$ $x^T_3 + a^F_{2,A} + t_{3\to2} \to \emptyset$ $x^F_4 + a^T_{2,A} + t_{4\to2} \to \emptyset$
2	$y^T_{1,N}$, $2a^T_{2,O}$, $y^T_{3,B}$ $t_{1\to1}$, $t_{3\to2}$ $y^F_{1,N}$, $2a^F_{2,O}$, $y^F_{3,B}$ $t_{2\to2}$, $t_{4\to3}$	$x^T_1 + y^T_{1,N} + t_{1\to1} \to \emptyset$ $x^T_2 + a^F_{2,O} + t_{2\to2} \to \emptyset$ $x^T_3 + a^F_{2,O} + t_{3\to2} \to \emptyset$ $x^T_4 + y^T_{3,B} + t_{4\to3} \to \emptyset$	**11**	$b^T_{1,A}$, $b^T_{2,A}$ $2u_1$, $2b^F_{1,A}$ $2b^F_{2,A}$, $2u_2$	$a^T_{1,A} + b^F_{1,A} + u_1 \to \emptyset$ $a^F_{1,A} + b^T_{1,A} + u_1 \to \emptyset$ $a^T_{2,A} + b^F_{2,A} + u_2 \to \emptyset$ $a^F_{2,A} + b^T_{2,A} + u_2 \to \emptyset$
3	$2b^T_{2,O}$, $2u_2$, $b^F_{2,O}$	$a^T_{2,O} + b^F_{2,O} + u_2 \to \emptyset$	**12**	$4v_1$, $4v_2$	*No Rules Apply*
4	$4v_2$	$u_2 + v_2 + v_2 \to \emptyset$	**13**	$2w_1$, $2w_2$	$w_1 + v_1 + v_1 \to \emptyset$ $w_2 + v_2 + v_2 \to \emptyset$
5	$2w_2$	$w_2 + v_2 + v_2 \to \emptyset$ $w_2 + a^T_{2,O} + b^T_{2,O} \to \emptyset$	**14**	$y^T_{1,A}$, $y^T_{2,A}$, $2t$ $y^F_{1,A}$, $y^F_{2,A}$	$b^F_{1,A} + y^T_{1,A} + t \to \emptyset$ $b^F_{2,A} + y^T_{2,A} + t \to \emptyset$
6	$y^T_{2,O}$, t, $y^F_{2,O}$	$b^T_{2,O} + y^F_{2,O} + t \to \emptyset$	**15**	x^T_1, x^T_2, $t_{1\to1}$ x^F_1, x^F_2, $t_{2\to2}$	$y^F_{1,A} + x^T_1 + t_{1\to1} \to \emptyset$ $y^F_{2,A} + x^T_2 + t_{2\to2} \to \emptyset$
7	$y^T_{2,O}$, r, $y^F_{2,O}$	$y^T_{2,O} + y^T_{2,O} + r \to \emptyset$	**16**	$2a^T_{1,O}$, $t_{1\to1}$ $2a^F_{1,O}$, $t_{2\to1}$	$x^F_1 + a^T_{1,O} + t_{1\to1} \to \emptyset$ $x^F_2 + a^T_{1,O} + t_{2\to1} \to \emptyset$
8	$2y^T_{2,O}$, $2y^F_{2,O}$	$y^F_{2,O} + y^F_{2,O} + y^F_{2,O} \to \emptyset$	**17**	$2b^T_{1,O}$, $2u_1$, $b^F_{1,O}$	$a^F_{1,O} + b^T_{1,O} + u_1 \to \emptyset$
9	x^T_1, x^T_2, x^T_3, x^T_4 $t_{1\to1}$, $t_{2\to3}$ x^F_1, x^F_2, x^F_3, x^F_4 $t_{2\to2}$, $t_{3\to4}$	$y^F_{1,N} + x^T_1 + t_{1\to1} \to \emptyset$ $y^T_{2,O} + x^F_2 + t_{2\to2} \to \emptyset$ $y^T_{2,O} + x^F_3 + t_{2\to3} \to \emptyset$ $y^F_{3,B} + x^T_4 + t_{3\to4} \to \emptyset$	**18**	$4v_1$	*No Rules Apply*
			19	$2w_1$	$w_1 + v_1 + v_1 \to \emptyset$
			20	$y^T_{1,O}$, t, $y^F_{1,O}$	$b^F_{1,O} + y^T_{1,O} + t \to \emptyset$
			21	x^T_1, $t_{1\to1}$ x^F_1	$y^F_{1,O} + x^T_1 + t_{1\to1} \to \emptyset$

Table 5: $(3, 0)$ rules and steps to compute the circuit in Figure 3c based on the indexing shown in Figure 3a. Note that as in other tables, the 'Steps' column shows the number and types of species being added at the beginning of that step.

To simulate circuits with $O(W^2)$ species, we also need to be able to reuse these input, output, and helper species. This can be accomplished by having unique species for each gate at a given depth level. Figure 3a shows a pattern the gates can be indexed in.

When reusing species, we incorporate a unique $t_{i\to j}$ species (different from the $t_{j\to i}$ species used in computing gates) for each gate at index i that converts the output species into an input species with the index j. Converting outputs into inputs is done for all gates at the same depth level. Table 6 shows the steps and rules needed to complete this process.

Fan Out. In order to perform a 2-fan out, we create a second copy of the output species that is fanning out. Table 7 shows the steps and rules needed to perform this duplication. After duplicating the output, the simulation continues as usual. All outputs at the same depth level can be fanned out at the same time using these two steps.

Complexity. The $t_{i\to j}$ approach results in, at most, W^2 unique species since $1 \leq i, j \leq W$. All other types of species either have $O(1)$ or $O(W)$ unique species. Therefore, a simulation of a circuit with a max fan-out of 2 requires $O(W^2)$ species.

All gates at a given depth level are evaluated at the same time, so a simulation of a circuit with a max fan-out of 2 requires $O(D)$ steps. Additionally, circuits are evaluated one depth level at a time. Thus, at most, a max width amount of input, output, and helper species are added at the same time. All of

Steps		Relevant Rules	Description
1	*Add* $\forall j \in f_i^{out} : x_j^T, x_j^F, t_{i\to j}$	$\forall j \in f_i^{out}$: $y_{i,g}^T + x_j^F + t_{i\to j} \to \emptyset$ $y_{i,g}^F + x_j^T + t_{i\to j} \to \emptyset$	$\forall j \in f_i^{out}$, convert $y_{i,g}^b$ output species into x_j^b input species.

Table 6: *(3, 0) rules for converting outputs into inputs per circuit level.*

Steps		Relevant Rules	Description
1	*Add* $y_{i,g}^T, y_{i,g}^F, r$	$y_{i,g}^T + y_{i,g}^T + r \to \emptyset$ $y_{i,g}^F + y_{i,g}^F + r \to \emptyset$	Flip output's bit (e.g. if species $y_{i,g}^T$ is present, then delete it and preserve $y_{i,g}^F$)
2	*Add* $2y_{i,g}^T, 2y_{i,g}^F$	$y_{i,g}^T + y_{i,g}^T + y_{i,g}^T \to \emptyset$ $y_{i,g}^F + y_{i,g}^F + y_{i,g}^F \to \emptyset$	Delete all copies of the negation of the initial input, and preserve the two copies of the input that were just added.

Table 7: *(3,0) rules and steps for 2-fan out.*

the input, output, and helper species from previous depth levels get deleted when progressing to the next depth level, so the simulation requires $O(W)$ volume. A constant number of species, steps, and volume are needed to perform a 2-fan out, so a 2-fan out operation does not affect the complexity. □

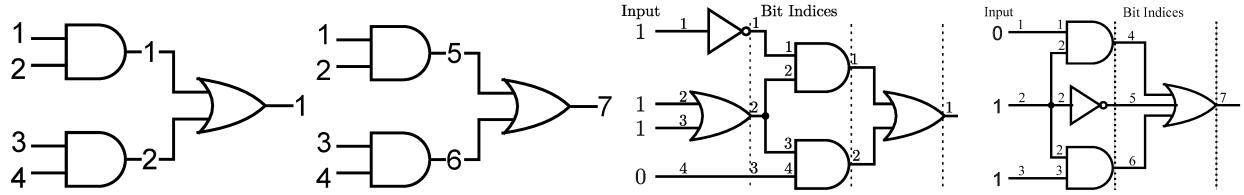
Lemma 3.2. *Threshold circuits (TC) with a max fan-out of 2 can be strictly computed by a step CRN with only* $(3, 0)$ *rules,* $O(G)$ *species,* $O(D)$ *steps, and* $O(W)$ *volume.*

Proof. Most of the rules, species, and steps used to compute a circuit with $O(W^2)$ species (Lemma 3.1) should also be used for this step CRN. The main difference is that there is an index for every gate in the circuit instead of limiting the indexes of these species by the max width. Figure 3b shows a pattern the gates can be indexed in. Also, we don't need the $t_{i\to j}$ species. This is because rules could overlap when reusing species, so the $t_{i\to j}$ species was used to make certain rules distinct and prevent the wrong reactions from occurring. However, each gate being represented by unique species eliminates the possibility of this error as every rule used to compute a gate will also be unique. So, in this step CRN, all instances where $t_{i\to j}$ species is used (including those in Section 3.1) are replaced by the generic t species.

Complexity. The $t_{i\to j}$ species that was the bottleneck for species in the step CRN discussed in Lemma 3.1 has been replaced by the t species. Therefore, the number of species is no longer upper bounded by $O(W^2)$. Instead, there are unique species for each gate, thus requiring $O(G)$ species. The differences discussed in this lemma do not affect the step or volume complexity determined in Lemma 3.1 ($O(D)$ and $O(W)$, respectively). □

Theorem 3.3. *Threshold circuits (TC) can be strictly computed by a step CRN with only* $(3, 0)$ *rules,* $O(\min(W^2, G \cdot F_{out}))$ *species,* $O(D \log F_{out})$ *steps, and* $O(W)$ *volume.*

Proof. Since the methods used in Lemmas 3.1 and 3.2 can only achieve a max fan-out of 2, a circuit with a larger fan-out (F_{out}) must be turned into a circuit with a max fan-out of 2. The max width does not change in the transformation process, but the total number of gates does. The transformation process creates a binary tree-like structure within the circuit, where the gate that is fanning out can be likened to the root, and the gates that the output is being inputted into can be likened to the leaves. Therefore, because a binary tree can have, at most, $2\ell - 1$ vertices, and an arbitrary fan out already has the "root" gate and "leaves" gates, the transformation process requires, at most, $\ell - 2$ more gates to construct the binary tree-like structure. Thus, the method requiring $O(G)$ species would require $O(GF_{out})$ species to simulate



(a) (b) (c) (d)

Figure 3: (a) Example indexing pattern of wires for the step CRN method using $O(W^2)$ species. (b) Example indexing pattern of wires for the step CRN method using $O(G)$ species. (c) Example circuit (with indexing) for Table 5. (d) Example circuit (with indexing) for Table 10.

a circuit with arbitrary fan-out. The most efficient method can be used for a given circuit, resulting in $O(\min(W^2, GF_{out}))$ species required to simulate a circuit.

Due to the binary tree-like structure made for gates with large fan out, the transformation process would increase the size of the depth from D to $D \log F_{out}$. Since the steps are dependent on the size of the depth, a circuit simulation would require $O(D \log F_{out})$ steps.

The volume of a circuit simulation does not differ between the two methods ($O(W)$ volume) nor is it affected by the transformation process. □

4 Threshold Circuits with (2, 0) and (2, 1) Catalyst Rules

Having established computation results with step CRNs using only true void rules, we now examine step CRNs whose rule-sets contain catalytic rules. These rulesets can either consist of only $(2, 1)$ rules or both $(2, 1)$ and $(2, 0)$ rules. Subsection 4.1 shows how the computation of logic gates is possible in step CRNs with just $(2, 0)$ or $(2, 1)$ rules. We then demonstrate with Theorem 4.3 how the system can compute TCs with $O(G)$ species, $O(D)$ steps, and $O(W)$ volume. Subsection 4.4 then shows that TCs can also be calculated (with more steps) with only the $(2, 1)$ catalyst rules.

4.1 Computing Logic Gates

Bit Representation and Indexing. The inputs of a binary gate are constructed the same as in Section 3.1. However, with catalysts, we slightly modify our indexing scheme. When fanning out, we do not split the output of the gate into input species with different indices because the catalyst rules remove the need to differentiate the input species. Let f_i^{in} be the set of all the indices of the inputs fanning into a gate at index i. Let f_i^{out} be the set of all the indices of the inputs fanning out of a gate at index i.

The output of a gate is represented by the species y_i^b or $y_{j\to i}^b$, where j is the index of the input bit and i is the index of the gate.

AND/OR/NOT Gate. Table 8 shows the general process to computing AND, OR, and NOT gates. To compute an AND gate, we add a single copy of the species representing a true output (y_i^T) and a species representing a false output for each input ($\forall j \in f_i^{in} : y_{j\to i}^F$). To compute an OR gate instead, we add a species representing a true output ($y_{j\to i}^T$) for each input and a single y_i^F species. To compute NOT gates, we add one copy of each output species (y_i^b). For every input into an AND/OR/NOT gate, a corresponding rule should be created to remove the output species of the gate with the opposite truth value to the input. If the output species has a unique $j \to i$ index, then only the input with the corresponding i can delete that output species.

These gates can also be computed with (2,1) catalyst rules by making the x_j^b species a catalyst. For example, the (2,0) rule $x_j^T + y_i^T \to \emptyset$ would be replaced by the (2,1) rule $x_j^T + y_i^T \to x_j^T$.

OR Example. Consider OR gate whose gate index is 1 with input bits 0 and 1. Here, $|f_i^{in}| = 2$, and the initial configuration consists of the species x_1^F and a x_2^T.

This gate can be computed in one step, following Table 8, by adding one $y_{1\to1}^T$, one $y_{2\to1}^T$, and one y_1^F species to the system. The species x_2^T and y_1^F delete each other. x_1^F and $y_{1\to1}^T$ are also removed together.

Gate Type	Step	Relevant Rules	Description
AND	*Add* y_i^T $\forall j \in f_i^{in} : y_{j\to i}^F$	$x_j^T + y_{j\to i}^F \to \emptyset$ $x_j^F + y_i^T \to \emptyset$	An input species with a certain truth value deletes the complement output species.
OR	*Add* y_i^F $\forall j \in f_i^{in} : y_{j\to i}^T$	$x_j^T + y_i^F \to \emptyset$ $x_j^F + y_{j\to i}^T \to \emptyset$	An input species with a certain truth value deletes the complement output species.
NOT	*Add* y_i^T y_i^F	$x_j^T + y_i^T \to \emptyset$ $x_j^F + y_i^F \to \emptyset$	The input and output species that share the same truth value delete each other.

Table 8: *(2, 0) rules for AND, OR, and NOT gates.*

Steps			Relevant Rules	Description
1	*Add*	$\lvert f_i^{in}\rvert \cdot a_i^T$ $\lvert f_i^{in}\rvert \cdot a_i^F$	$\forall j \in f_i^{in}$: $x_j^T + a_i^F \to \emptyset$ $x_j^F + a_i^T \to \emptyset$	$\forall j \in f_i^{in}$, convert x_j^b input species into a_i^b species.
2	*Add*	$\lfloor \lvert f_i^{in}\rvert /2 \rfloor \cdot b_i^T$ $\lfloor \lvert f_i^{in}\rvert /2 \rfloor \cdot b_i^F$	$a_i^T + b_i^F \to \emptyset$ $a_i^F + b_i^T \to \emptyset$	Adding $\lfloor \lvert f_i^{in}\rvert /2 \rfloor$ amounts of b_i^T and b_i^F species will delete all of the minority species, leaving some amount of the majority species remaining.
3	*Add*	y_i^T y_i^F	$a_i^T + y_i^F \to \emptyset$ $a_i^F + y_i^T \to \emptyset$	Convert a_i^b into the proper output species (y_i^b).

Table 9: *(2, 0) rules for majority gates.*

Only the species $y_{2\to 1}^T$ remains, which represents the intended "true" output of the OR gate.

Majority Gate. The general process of computing a majority gate is shown at Table 9. To compute a majority gate, all input species are converted into a new species a_i^b (step 1). The species retains the same truth value b as the original input and has the gate index i. If the number of species fanning into the majority gate is even, an extra *false* input species is added in. The species b_i^b is then introduced, which computes the majority operation across all existing species. Any species that represent the minority inputs are deleted (step 2). The species remaining afterwards are then converted into the correct output species (step 3).

4.2 Examples

With the computation of individual gates demonstrated in our system, we now expand these features to computing entire circuits. We begin with a simple example in Figure 3d to show the concepts before giving the general construction.

Our example circuit has three inputs: x_1, x_2, and x_3. In the first layer, x_2 is fanned out three times. One is fanned into an AND gate with x_1, another fanned into a NOT gate, and the other fanned into an AND gate with x_3. Finally, at the next depth level, the output of all three gates are fanned into an OR gate, whose output is the final circuit output.

Table 10 shows how to compute the circuit in Figure 3d. The primary inputs of the circuit in Figure 3d are represented by the species in the initial configuration. Steps 1-5 fan out the second primary input, convert the output species (y_n^b) into input species (x_n^b), and delete excess species. Step 6 computes the gates at the first depth level. Steps 7-11 convert the output species into input species and deletes excess species. Step 12 computes the final gate. Steps 13-17 delete excess species and converts the output of the final gate into an input species that represents the solution to the circuit (x_7^T).

Initial Configuration:							
y_1^F y_2^T y_3^T							
Steps			**Relevant Rules**		**Steps**		**Relevant Rules**
1	*Add*	d_x	*No Rules Apply*	**8**	*Add*	d_x	$d_x + d_x \to \emptyset$
2	*Add*	d_x	$d_x + d_x \to \emptyset$	**9**	*Add*	x_4^T, x_4^F x_5^T, x_5^F x_6^T, x_6^F	$y_{1\to4}^F + x_4^T \to y_{1\to4}^F$ $y_5^F + x_5^T \to y_5^F$ $y_6^T + x_6^F \to y_6^T$
3	*Add*	x_1^T, x_1^F $3x_2^T$, $3x_2^F$ x_3^T, x_3^F	$y_1^F + x_1^T \to y_1^F$ $y_2^T + x_2^F \to y_2^T$ $y_3^T + x_3^F \to y_3^T$	**10**	*Add*	d_y	$y_{1\to4}^F + d_y \to d_y$ $y_5^F + d_y \to d_y$ $y_6^T + d_y \to d_y$
4	*Add*	d_y	$y_1^F + d_y \to d_y$ $y_2^T + d_y \to d_y$ $y_3^T + d_y \to d_y$	**11**	*Add*	d_y	$d_y + d_y \to \emptyset$
5	*Add*	d_y	$d_y + d_y \to \emptyset$	**12**	*Add*	$y_{4\to7}^T$, $y_{5\to7}^T$ $y_{6\to7}^T$, y_7^F	$x_4^F + y_{4\to7}^T \to \emptyset$ $x_5^F + y_{5\to7}^T \to \emptyset$ $x_6^T + y_7^F \to \emptyset$
6	*Add*	y_4^T, $y_{1\to4}^F$ y_5^T, $y_{2\to4}^F$ y_5^F, $y_{2\to6}^F$ y_6^T, $y_{3\to6}^F$	$x_1^F + y_4^T \to \emptyset$ $x_2^T + y_{2\to4}^F \to \emptyset$ $x_2^T + y_5^T \to \emptyset$ $x_2^T + y_{2\to6}^F \to \emptyset$ $x_3^T + y_{3\to6}^F \to \emptyset$	**13**	*Add*	d_x	*No Rules Apply*
				14	*Add*	d_x	$d_x + d_x \to \emptyset$
				15	*Add*	x_7^T, x_7^F	$y_{6\to7}^T + x_7^F \to y_{6\to7}^T$
				16	*Add*	d_y	$y_{6\to7}^T + d_y \to d_y$
7	*Add*	d_x	*No Rules Apply*	**17**	*Add*	d_y	$d_y + d_y \to \emptyset$

Table 10: *(2, 0) and (2, 1) rules and steps to compute the circuit in Figure 3d with Figure 3b's indexing.*

4.3 Computing Circuits with (2,0) Void and (2,1) Catalyst Rules

Theorem 4.1. *Threshold circuits (TC) can be strictly computed with* $(2, 0)$ *void rules and* $(2, 1)$ *catalyst rules,* $O(G)$ *species,* $O(D)$ *steps, and* $O(W)$ *volume.*

Proof. With only $(2, 0)$ rules, there is no known way to perform fan-outs without introducing, at most, an exponential count of species at certain steps. This makes it impossible to strictly compute circuits with only $(2, 0)$ rules and results in a large volume. The use of $(2, 1)$ catalyst rules enables the step CRN to compute with arbitrary fan-out without an increase in species count, as well as deleting all species that are no longer needed. This allows for strict computation and decreases the volume of the step CRN to a polynomial size.

The initial configuration of this step CRN should consist of a y_n^b species with the appropriate indexing and truth values for each primary input. Section 4.1 explains how to compute TC gates. To run the circuit, we must convert the output species into input species. In addition, this step CRN uses d_x and d_y as *deleting* species. Table 11 shows the steps and rules needed to perform arbitrary fan-out for a gate. All outputs at the same depth level can be fanned out at the same time.

Complexity: Having a constant amount of unique species represent each gate in a circuit and a constant number of helper species results in $O(G)$ species. All gates at a given depth level are computed at the same time in a constant number of steps. Thus, the circuit simulation requires $O(D)$ steps. This step CRN only needs to introduce a constant number of species to compute each gate, and it deletes all species no longer needed after computing all gates at a given depth level. Thus, the step CRN requires $O(W)$ volume. □

4.4 Computing Circuits with (2,1) Catalyst Rules

It's worth noting that (2,1) catalyst rules are able to compute TCs on their own. The main drawback is that there is no known way to directly compute majority gates without $(k \geq 2, 0)$ void rules. Thus, any majority gate must be computed using AND, OR, and NOT gates when using only catalyst rules. Furthermore,

Steps			Relevant Rules	Description
1	*Add*	d_x	$\forall n \in \{1, \cdots, G\}$: $\forall b \in \{T, F\}$ $d_x + x_n^b \to d_x$ $d_x + a_n^b \to d_x$ $d_x + b_n^b \to d_x$	Delete all input species (x_n^b) and helper species that are no longer needed.
2	*Add*	d_x	$d_x + d_x \to \emptyset$	Remove deleting species d_x.
3	*Add*	$\lvert f_i^{out}\rvert \cdot x_i^T$ $\lvert f_i^{out}\rvert \cdot x_i^F$	$y_i^T + x_i^F \to y_i^T$ $y_i^F + x_i^T \to y_i^F$ $\forall j \in f_i^{in}$: $y_{j\to i}^T + x_i^F \to y_{j\to i}^T$ $y_{j\to i}^F + x_i^T \to y_{j\to i}^F$	Add species representing true and false inputs and delete the species that are the complement of the output. A single output species can assign the truth value for as many input species as needed.
4	*Add*	d_y	$\forall n \in \{1, \cdots, G\}$: $d_y + y_n^T \to d_y$ $d_y + y_n^F \to d_y$ $\forall j \in f_i^{in}$: $d_y + y_{j\to i}^T \to d_y$ $d_y + y_{j\to i}^F \to d_y$	Delete all output species (y_n^b) that no longer needed.
5	*Add*	d_y	$d_y + d_y \to \emptyset$	Remove deleting species d_y.

Table 11: *(2, 0) and (2, 1) rules and steps for a gate with arbitrary fan out.*

deleting species that are no longer needed is slightly more convoluted with (2,1) rules compared to pure void rules.

Corollary 4.2. *Threshold circuits (TC) can be strictly computed with only* $(2, 1)$ *catalyst rules,* $O(G)$ *species,* $O(D \log F_{maj})$ *steps, and* $O(W)$ *volume.*

Proof. Section 4.1 explains how to compute AND/OR/NOT gates using (2,0) rules, and they can easily be changed to (2,1) rules by making the x_j^b species a catalyst. The method used to perform arbitrary fan out in Theorem 4.1 can be slightly modified to function with only (2,1) rules. Table 12 demonstrates how this can be done. A special property of using (2,1) rules to compute gates is that the counts of the species being added are flexible. This is not the case when gates are computed with pure void rules, as it is necessary for the counts of certain species to be precise. For example, while Step 3 in Table 11 and Step 5 in Table 12 are functionally equivalent steps, they have different computing requirements. When computing with (2,0) rules, we need exactly $|f_i^{out}|$ amount of x_i^b species by the end of that step. On the other hand, when computing with (2,1) rules, we only need one copy of x_i^b, and even if multiple copies of that species were added, it would not have a significant impact on the computation of the circuit.

Complexity: The techniques used to compute circuits are functionally equivalent to the ones used in Theorem 4.1, so the upper bound of species and volume remain the same, that is, $O(G)$ and $O(W)$, respectively.

Since majority gates must be computed using AND and OR gates when using only (2,1) rules, the depth of the circuit must increase. The conversion of a majority gate to AND/OR/NOT gates can be achieved with $O(n)$ gates and $O(\log n)$ depth where n is the number of input bits of the majority gate [35] (any symmetric boolean function has a circuit of depth $O(\log n)$ and size $O(n)$ for n bits). Thus, the maximum number of steps needed to compute a circuit would be $O(D \log F_{maj})$, where F_{maj} is the maximum fan-in of any majority gate in the circuit. □

Steps		Relevant Rules	Description
1	*Add* d'_x	$d'_x + d''_x \to d'_x$	Deleting species d''_x makes it possible for species d_x to exist in the next step without complications.
2	*Add* d_x	$d_x + d'_x \to d_x$ $\forall n \in \{1, \cdots, G\}$: $\forall b \in \{T, F\}$ $d_x + x_n^b \to d_x$ $d_x + a_n^b \to d_x$ $d_x + b_n^b \to d_x$	Deleting species d'_x makes it possible for species d''_x to exist in the next step without complications. Delete all input species (x_n^b) and helper species that are no longer needed.
3	*Add* d''_x	$d_x + d''_x \to d''_x$	Removes deleting species d_x.
4	*Add* d'_y	$d'_y + d''_y \to d'_y$	Deleting species d''_y makes it possible for species d_y to exist in the next step without complications.
5	*Add* x_i^T x_i^F	$y_i^T + x_i^F \to y_i^T$ $y_i^F + x_i^T \to y_i^F$ $\forall j \in f_i^{in}$: $y_{j\to i}^T + x_i^F \to y_{j\to i}^T$ $y_{j\to i}^F + x_i^T \to y_{j\to i}^F$	Add species representing true and false inputs and delete the species that are the complement of the output. A single output species can assign the truth value for as many input species as needed.
6	*Add* d_y	$d_y + d'_y \to d_y$ $\forall n \in \{1, \cdots, G\}$: $d_y + y_n^T \to d_y$ $d_y + y_n^F \to d_y$ $\forall j \in f_i^{in}$: $d_y + y_{j\to i}^T \to d_y$ $d_y + y_{j\to i}^F \to d_y$	Deleting species d'_y makes it possible for species d''_y to exist in the next step without complications. Delete all output species (y_n^b) that are no longer needed.
7	*Add* d''_y	$d_y + d''_y \to d''_y$	Remove deleting species d_y.

Table 12: *(2, 1) rules and steps for a gate with arbitrary fan out.*

5 Lower Bounds and Hardness

In this section, we prove negative results for computing with step CRNs. First, we show there exists a family of functions that require a logarithmic number of steps to compute. Then, we show hardness of verifying whether a step CRN properly computes a given function.

5.1 Step Lower Bound for Controlled NOT

CNOT. The Controlled NOT gate is a 2-bit input and 2-bit output gate taking inputs X and Y, and outputting X and $X \oplus Y$. In other words, the gate flips Y if X is true.

k-CNOT. We generalize this to a Controlled k-NOT gate. This is a $(k+1)$-bit gate with inputs $X, Y_1, \ldots, Y_k$. The Y bits all flip if X is true. We choose this function since it has the property that changing 1 bit of the input changes a large number of output bits.

Configuration Distance. Recall configurations are defined as vectors. For two configurations c_0, c_1, we say the distance between them is $||c_0 - c_1||_1$, i.e., the sum of the absolute value of each entry in $c_0 - c_1$.

Lemma 5.1. *Let r be a positive integer parameter. For all step CRNs Γ with void rules of size $(r_1, 0)$ with $r_1 \leq r$ and pairs of initial configurations c_T and c_F with distance 2 and equal volume, for any configuration*

c_{Ts} terminal in the step s from c_T, there exists a configuration c_{Fs} terminal in step s from c_F such that the distance between c_{Ts} and c_{Fs} is $O(r^s)$.

Proof. Let T be the species that is in configuration c_T and not in c_F. Similarly, define the species F to be the species in configuration c_F and not in c_T. Consider the reaction sequence R_T starting from c_T and ending in c_{T1}. All but one reaction in R_T can be applied to c_F. Consider the reaction sequence R_F that differs from R_T by two reactions. The first is the reaction in R_T that consumes T and $r-1$ other species, and the second is a reaction that consumes F and $r-1$ other species. Applying R_F results in a configuration c_{F1} that differs from c_{T1} by at most $2r$ species.

Now, assume there are two initial configurations in step s with distance $2r^{s-1}$ away from each other. In the base case, each different species between the configurations can be used in at most one rule, thus the species can only propagate $r-1$ additional changes in the next terminal configuration. This results in a difference of $2rr^{s-1} = 2r^s$ in the s stage. □

The configuration distance between two output configurations is related to the Hamming distance of the output strings they represent. Lemma 5.1 can be used to get a get a logarithmic lower bound for the number of steps required when we fix our rule size to be a constant.

Theorem 5.2. *For all constants r, any CRN that strictly computes a k-CNOT gate with rules of size $(r_1, 0)$ satisfying $r_1 \leq r$ requires $\Omega(\log k)$ steps.*

Proof. Flipping only the X bit of the the input changes all $k+1$ output bits. It follows that in order to compute a k-CNOT, we must have at least k distance between the two final configurations even when starting from configurations with distance 1. We can also assume these have the same volume since by changing a bit value we are only changing which species we add. With Lemma 5.1, we get the following inequality and must compute s.

$$k \leq 2r^s \qquad \rightarrow \qquad \log k \leq 2s \log r \qquad \rightarrow \qquad \frac{\log k}{2 \log r} \leq s$$

Since r is a constant we get an asymptotic bound of $s = \Omega(\log k)$. □

We also note the k-CNOT can be computed by k XOR gates in parallel. This implies this lower bound does not hold with catalytic reactions either as Theorem 4.1 shows this can be computed in $O(1)$ steps or without the input-strict requirement. This is because increasing the fan-out of the X bit does not incur a cost in the number of steps in both of these generalizations. Plugging this XOR circuit into Theorem 3.3 gives a bound of $\Theta(\log k)$ steps showing the construction is optimal for some circuits.

5.2 Function Verification Hardness

We have established that void step CRNs can simulate Boolean circuits. We now discuss the complexity of the computational problem of determining if a given (void) step CRN does compute a given function. Specifically, we consider the following decision problem:

(Strict Function Verification): Given a step CRN $C_S = (S, X, Y)$ and a Boolean function $f(\cdot)$[1] where $f(x_1, \cdots, x_n) = y_1 : \{0,1\}^n \rightarrow \{0,1\}$, decide if C_S computes Boolean function $f(\cdot)$. In particular, let $f_0(x_1, \cdots, x_n) = false$, which is false for all inputs.

Theorem 5.3 shows that strict function verification in a step CRN system with void rules is coNP-hard, and coNP membership for this problem is shown in Theorem 5.4.

Theorem 5.3. *It is coNP-hard to determine if a given a $\mathcal{O}(1)$-step CRN $C_S = (S, X, Y)$ with $(3, 0)$ rules computes the boolean function $f_0(x_1, \cdots, x_n)$.*

[1]We assume that $f(\cdot)$ is given in the form of a circuit c_f. We leave as future work the complexity of other representations such as a truth table.

Proof. The decision problem 3SAT is a classical NP-complete problem. Its complementary problem, which is coNP-complete, is to decide if the conjunction of several clauses with three literals cannot be satisfied. We also reduce from the special case of 3SAT where each variable appears at most 4 times, shown to be NP-complete in [41]. Let $F(x_1, \cdots, x_n) = C_1 \wedge C_2 \wedge \cdots C_m$ be an instance for a 3SAT problem, where each C_i is a clause that has at most three literals, for example, $C_1 = (x_1 \vee \overline{x_2} \vee x_3)$. The function $F(.)$ can be computed by a boolean circuit of constant depth. Checking if $F(x_1, \cdots, x_n) = f_0(x_1, \cdots, x_n)$ for all $(x_1, \cdots, x_n)$ is the complementary problem for 3SAT.

It follows from Theorem 3.3, which shows $F(.)$ can be simulated by a step CRN $C_S = (S, X, Y)$ with $(3, 0)$ rules. The number of steps of the CRN is $\mathcal{O}(D \log F_o ut)$. The depth of the circuit is constant since all clauses can be computed in parallel and the gates handle arbitary fan-in. The fanout is also constant because each variable only appears 4 times. □

Theorem 5.4. *Determining if a given a s-step CRN $C_S = (S, X, Y)$ with $(r, 0)$ rules computes the boolean function $f_0(x_1, \cdots, x_n)$ is in coNP.*

Proof. This problem can be solved by a polynomial time non-deterministic algorithm which does the following. Pick a string b of length n and compute $f_0(b) = Y$. Then convert b to the initial configuration $X(b)$. Guess a sequence of s terminal configurations $c_1, \ldots, c_s$ where c_i is a terminal configuration in the ith step. To verify this, call the NP algorithm for reachability in Volume Decreasing CRNs from [1] to verify each configuration is reachable in the correct step. If the final configuration c_s does not represent Y then reject. □

Theorem 5.5. *It is coNP-Complete to determine if a given a $O(1)$-step CRN $C_S = (S, X, Y)$ with $(3, 0)$ rules computes the boolean function $f_0(x_1, \cdots, x_n)$.*

Proof. Follows from Theorems 5.3 and 5.4. □

6 Conclusions and Future Work

We have proposed the step CRN model, a natural augmentation to the CRN model, and shown that void rule CRNs, a simple but computationally weak class of CRNs, become capable of efficiently simulating Threshold Circuits under this extension. We have shown this holds even when limited to $(3, 0)$ reactions, and further shown that bi-molecular reactions are equally powerful if permitted access to catalytic reactions. We also show that the step augmentation is fundamentally needed: without access to a super-constant number of steps, such computation is impossible. Finally, we utilize our positive results to show coNP-completeness for the problem of deciding if a given step CRN computes a given function.

The step CRN model presented in this work, along with our results, naturally lead to a number of additional promising research directions. A small sample of these are:

- Lower Bounds and Catalysts: We conjecture $(3, 0)$ void rules are the smallest size rules for strictly simulating TC without the use of a catalyst. Further, we show that more than a constant number of steps are required for circuit computation for non-catalytic void rules, but is it true with a catalyst?
- Robustness: While void rules potentially offer a simpler path to experimental feasibility and scalability, some of our techniques require precise counts of species to be added at different steps of the computational process. Such precision is a hurdle to experimental implementation. Thus, it is interesting to consider to what extent these results can be made robust to approximate counts (or have a lower-bound).
- Reachability: The *reachability* problem of determining if a given configuration is reachable from an initial configuration is well-studied in CRNs and other computational models. We showed that steps allow for greater computational power with void rules. How does the addition of steps affect the reachability problem, and specifically, what is the complexity when relating void rules, catalysts, rule size, and number of steps?

- General Staged CRNs: We explored a simple scheme for including separate stages into the CRN model by simply adding new species at each step. A more general modelling could include multiple separate bins that may be mixed or split together over a sequence of stages. Formalizing such a model and exploring the added power of this generalization is an interesting direction for future work.

References

[1] Robert M. Alaniz, Bin Fu, Timothy Gomez, Elise Grizzell, Andrew Rodriguez, Robert Schweller, and Tim Wylie. Reachability in restricted chemical reaction networks, 2022. arXiv:2211.12603.

[2] Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distribed Computing*, 18(4):235–253, mar 2006. doi:10.1007/s00446-005-0138-3.

[3] Dana Angluin, James Aspnes, and David Eisenstat. A simple population protocol for fast robust approximate majority. *Distributed Computing*, 21:87–102, 2008.

[4] Dana Angluin, James Aspnes, David Eisenstat, and Eric Ruppert. The computational power of population protocols. *Distributed Computing*, 2007.

[5] Rutherford Aris. Prolegomena to the rational analysis of systems of chemical reactions. *Archive for Rational Mechanics and Analysis*, 19(2):81–99, jan 1965. doi:10.1007/BF00282276.

[6] Rutherford Aris. Prolegomena to the rational analysis of systems of chemical reactions ii. some addenda. *Archive for Rational Mechanics and Analysis*, 27(5):356–364, jan 1968. doi:10.1007/BF00251438.

[7] Adam Arkin and John Ross. Computational functions in biochemical reaction networks. *Biophysical journal*, 67(2):560–578, 1994.

[8] Ari. Aviram. Molecules for memory, logic, and amplification. *Journal of the American Chemical Society*, 110(17):5687–5692, Aug 1988. doi:10.1021/ja00225a017.

[9] Stefan Badelt, Seung Woo Shin, Robert F Johnson, Qing Dong, Chris Thachuk, and Erik Winfree. A general-purpose crn-to-dsd compiler with formal verification, optimization, and simulation capabilities. In *International conference on DNA-based computers*, pages 232–248. Springer, 2017.

[10] Z Beiki, Z Zare Dorabi, and Ali Jahanian. Real parallel and constant delay logic circuit design methodology based on the dna model-of-computation. *Microprocessors and Microsystems*, 61:217–226, 2018.

[11] Luca Cardelli and Attila Csikász-Nagy. The cell cycle switch computes approximate majority. *Scientific reports*, 2(1):656, 2012.

[12] Luca Cardelli, Marta Kwiatkowska, and Max Whitby. Chemical reaction network designs for asynchronous logic circuits. *Natural computing*, 17:109–130, 2018.

[13] Luca Cardelli, Mirco Tribastone, and Max Tschaikowski. From electric circuits to chemical networks. *Natural Computing*, 19:237–248, 2020.

[14] Cameron Chalk, Eric Martinez, Robert Schweller, Luis Vega, Andrew Winslow, and Tim Wylie. Optimal staged self-assembly of general shapes. *Algorithmica*, 80:1383–1409, 2018.

[15] Ho-Lin Chen, David Doty, and David Soloveichik. Deterministic function computation with chemical reaction networks. *Natural computing*, 13(4):517–534, 2014.

[16] Sonya C Cirlos, Timothy Gomez, Elise Grizzell, Andrew Rodriguez, Robert Schweller, and Tim Wylie. Simulation of multiple stages in single bin active tile self-assembly. In *International Conference on Unconventional Computation and Natural Computation*, pages 155–170. Springer, 2023.

[17] Matthew Cook, David Soloveichik, Erik Winfree, and Jehoshua Bruck. *Programmability of Chemical Reaction Networks*, pages 543–584. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. doi:10.1007/978-3-540-88869-7_27.

[18] Neil Dalchau, Harish Chandran, Nikhil Gopalkrishnan, Andrew Phillips, and John Reif. Probabilistic analysis of localized dna hybridization circuits. *ACS synthetic biology*, 4(8):898–913, 2015.

[19] Erik D Demaine, Sarah Eisenstat, Mashhood Ishaque, and Andrew Winslow. One-dimensional staged self-assembly. *Natural Computing*, 12(2):247–258, 2013.

[20] Samuel J Ellis, Titus H Klinge, and James I Lathrop. Robust chemical circuits. *Biosystems*, 186:103983, 2019.

[21] Abeer Eshra and Ayman El-Sayed. An odd parity checker prototype using dnazyme finite state machine. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 11(2):316–324, 2013.

[22] Daoqing Fan, Yongchao Fan, Erkang Wang, and Shaojun Dong. A simple, label-free, electrochemical dna parity generator/checker for error detection during data transmission based on "aptamer-nanoclaw"-modulated protein steric hindrance. *Chemical Science*, 9(34):6981–6987, 2018. doi:10.1039/C8SC02482K.

[23] Daoqing Fan, Jun Wang, Jiawen Han, Erkang Wang, and Shaojun Dong. Engineering dna logic systems with non-canonical dna-nanostructures: Basic principles, recent developments and bio-applications. *Science China Chemistry*, 65(2):284–297, 2022.

[24] Allen Hjelmfelt, Edward D Weinberger, and John Ross. Chemical implementation of neural networks and turing machines. *Proceedings of the National Academy of Sciences*, 88(24):10983–10987, 1991.

[25] Hua Jiang, Marc D Riedel, and Keshab K Parhi. Digital logic with molecular reactions. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 721–727. IEEE, 2013.

[26] Richard M. Karp and Raymond E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3(2):147–195, 1969. doi:https://doi.org/10.1016/S0022-0000(69)80011-5.

[27] Matthew R Lakin, David Parker, Luca Cardelli, Marta Kwiatkowska, and Andrew Phillips. Design and analysis of dna strand displacement devices using probabilistic model checking. *Journal of the Royal Society Interface*, 9(72):1470–1485, 2012.

[28] Yu-Chou Lin and Jie-Hong R Jiang. Mining biochemical circuits from enzyme databases via boolean reasoning. In *Proceedings of the 39th International Conference on Computer-Aided Design*, pages 1–9, 2020.

[29] David C Magri. A fluorescent and logic gate driven by electrons and protons. *New Journal of Chemistry*, 33(3):457–461, 2009.

[30] Shay Mailloux, Nataliia Guz, Andrey Zakharchenko, Sergiy Minko, and Evgeny Katz. Majority and minority gates realized in enzyme-biocatalyzed systems integrated with logic networks and interfaced with bioelectronic systems. *The Journal of Physical Chemistry B*, 118(24):6775–6784, Jun 2014. doi:10.1021/jp504057u.

[31] Dandan Mo and Darko Stefanovic. Iterative self-assembly with dynamic strength transformation and temperature control. In *DNA Computing and Molecular Programming: 19th International Conference, DNA 19, Tempe, AZ, USA, September 22-27, 2013. Proceedings 19*, pages 147–159. Springer, 2013.

[32] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Rheinisch-Westfälischen Institutes für Instrumentelle Mathematik an der Universität Bonn, 1962.

[33] Lulu Qian and Erik Winfree. Scaling up digital circuit computation with dna strand displacement cascades. *science*, 332(6034):1196–1201, 2011.

[34] Lulu Qian and Erik Winfree. A simple dna gate motif for synthesizing large-scale circuits. *Journal of The Royal Society Interface*, 8(62):1281–1297, Feb 2011. doi:10.1098/rsif.2010.0729.

[35] Igor Sergeevich Sergeev. Upper bounds for the formula size of symmetric boolean functions. *Russian Mathematics*, 58:30–42, 2014.

[36] David Soloveichik, Matthew Cook, Erik Winfree, and Jehoshua Bruck. Computation with finite stochastic chemical reaction networks. *natural computing*, 7(4):615–633, 2008.

[37] David Soloveichik, Georg Seelig, and Erik Winfree. Dna as a universal substrate for chemical kinetics. *Proceedings of the National Academy of Sciences*, 107(12):5393–5398, 2010.

[38] Chris Thachuk and Anne Condon. Space and energy efficient computation with dna strand displacement systems. In *International Workshop on DNA-Based Computers*, 2012.

[39] Chris Thachuk, Erik Winfree, and David Soloveichik. Leakless dna strand displacement systems. In *DNA Computing and Molecular Programming: 21st International Conference, DNA 21, Boston and Cambridge, MA, USA, August 17-21, 2015. Proceedings 21*, pages 133–153. Springer, 2015.

[40] Anupama J Thubagere, Chris Thachuk, Joseph Berleant, Robert F Johnson, Diana A Ardelean, Kevin M Cherry, and Lulu Qian. Compiler-aided systematic construction of large-scale dna strand displacement circuits using unpurified components. *Nature Communications*, 8(1):1–12, 2017.

[41] Craig A Tovey. A simplified np-complete satisfiability problem. *Discrete applied mathematics*, 8(1):85–89, 1984.

[42] Marko Vasić, David Soloveichik, and Sarfraz Khurshid. Crn++: Molecular programming language. *Natural Computing*, 19:391–407, 2020.

[43] Boya Wang, Chris Thachuk, Andrew D Ellington, Erik Winfree, and David Soloveichik. Effective design principles for leakless strand displacement systems. *Proceedings of the National Academy of Sciences*, 115(52):E12182–E12191, 2018.

[44] Erik Winfree. Chemical reaction networks and stochastic local search. In *DNA Computing and Molecular Programming: 25th International Conference, DNA 25, Seattle, WA, USA, August 5–9, 2019, Proceedings 25*, pages 1–20. Springer, 2019.

[45] Wei Xiao, Xinjian Zhang, Zheng Zhang, Congzhou Chen, and Xiaolong Shi. Molecular full adder based on dna strand displacement. *IEEE Access*, 8:189796–189801, 2020. doi:10.1109/ACCESS.2020.3031221.

[46] David Yu Zhang and Georg Seelig. Dynamic dna nanotechnology using strand-displacement reactions. *Nature chemistry*, 3(2):103–113, 2011.