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, 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, Edinburg, TX 78539-2999, USA
[2] Massachusetts Institute of Technology, Cambridge, MA 02139, USA
[3] University of Texas Dallas, Richardson, TX 75080-3021, USA

Abstract. We introduce a new model of *step* Chemical Reaction Networks (step CRNs), motivated by the step-wise 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 [6,7]. 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 [13], such as Vector Addition Systems (VASs) [19] and Petri-Nets [21]. Further, Population Protocols [3] 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 (and VASs, Petri-Nets) motivated by the desire to reflect standard laboratory and medical practices (and multi-step distributed processes). The *Step* CRN model 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

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

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 powerful, computationally universal class of *Threshold Circuits* (TC) using only *void* rules, a class of rules that are provably weak without a step augmentation (Theorem 3).

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 [26]. However, implementing this level of generality requires sophisticated, and large, DNA complexes that incur practical errors and constitute one of the primary hurdles limiting the scalable implementation of molecular computing schemes [11,28].

Here, we focus on *void* rules, a class of rules that utilize only the *removal* feature of CRN rules. Note that by removal, we simply mean that both elements can no longer be used, which may be that they become some species that is easily filtered. While simpler, the class of pure void rulse is unable to compute even simple functions such as the CNOT gate (Theorem 3). We show that void rules become computationally powerful in the step model with just tri-molecular or bi-molecular interactions. Specifically, that TCs can be simulated with void rules using a number of steps linear in the circuit's depth.

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 are the ability to simulate the class of 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 of TC with void rules of size $(3, 0)$ 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 $(3, 0)$ circuits. Finally, we show that it is coNP-complete to know whether a function can be strictly simulated by a step CRN system.

1.1 Previous Work

Computation in Chemical Reaction Networks. Stochastic Chemical Reaction Networks are only Turing-complete with the possibility for error [25] while error-free stochastic Chemical Reaction Networks can compute precisely the set

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 1
$(2,0)(2,1)$	$O(G)$	$O(D)$	Strict	**TC** Circuits	Theorem 2
$(2,1)$	$O(G)$	$O(D \log F_{maj})$	Strict	**TC** Circuits	Corollary 1
$(c,0)$	any	$\Omega(\log k)$	Strict	k-CNOT	Theorem 3

Strict Function Verification			
Rules	**Steps**	**Complexity**	**Ref**
$(3,0)$	$O(1)$	coNP-complete	Theorem 6

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, F_{maj} is the max fan-in of majority gates, and **TC** is Threshold Circuits. The k-CNOT is a k fan-in generalization of a Controlled NOT gate. Rule $(c,0)$ means any size with integer constant $c > 0$.

of semilinear functions [5,12]. CRNs have also recently been shown to be experimentally viable through DNA Strand Displacement (DSD) systems [26] with several CRN to DSD compilers now existing.

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 [9] and an initial presentation of circuits with CRNs in 1991 [17]. Since then, the area has been extensively studied in CRNs and related models [8,10,13,18,22,23]. Numerous gates have been built experimentally and proposed theoretically such as the AND [23,30], OR [14,23], NOT [10], XOR [10,30], NAND [13,29], NOR [10], Parity [16], and Majority [4,20]. 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 [24].

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. 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 [29].

Mixing Systems. Another generalization of CRNs that is closely related to the step model is I/O CRNs [15], 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.

2 Preliminaries

Basics. Let $\Lambda = \{\lambda_1, \lambda_2, \cdots, \lambda_{|\Lambda|}\}$ denote some ordered alphabet of *species*. A configuration C over Λ is a length-$|\Lambda|$ vector of integers where the i^{th} entry $C[i]$ denotes the number of copies of species λ_i. A *rule* or *reaction* is represented as an ordered pair of configuration vectors $R = (R_r, R_p)$. 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 $C \rightarrow^1_\Gamma C'$, a relation denoting that C can transition to C' by way of a single rule application from Γ. We further use the notation $C \rightarrow^*_\Gamma C'$ to signify the transitive closure of $\rightarrow^1_\Gamma$ 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 $R = (R_r, R_p)$: $R_r[1]\lambda_1 + \cdots + R_r[|\Lambda|]\lambda_{|\Lambda|} \rightarrow R_p[1]\lambda_1 + \cdots + R_p[|\Lambda|]\lambda_{|\Lambda|}$. 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 $2H + O \rightarrow W$.

Definition 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 I and CRN (Λ, Γ) are together 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 $TERM_{I,\Lambda,\Gamma}$.

Definition 2 (Void rules). *A rule $R = (R_r, R_p)$ is a* void *rule if the application vector $R_p - R_r$ has no positive entries and at least one negative entry. We say its a true void rule if the R_p vector is the* 0 *vector, and catalytic otherwise.*

Definition 3 (Volume and Rule Size). *The* size/volume *of a configuration vector C is* $\texttt{volume}(C) = \sum_{i=1}^{|\Lambda|} C[i]$. *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 $i = 3$ and bimolecular if $i = 2$.*

2.1 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 $((\Lambda, \Gamma), (s_0, s_1, s_2, \cdots, s_{k-1}))$, 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 to add at each step. Figure 1 shows an example system.

Given a step CRN, we define the set of reachable configurations after each sequential step. To start off, let REACH_1 be the set of reachable configurations of (Λ, Γ) with initial configuration s_0, which we refer to as the set of configurations reachable *after step 1*. Let TERM_1 be the subset of configurations in REACH_1 that are terminal. Note that after a single step we have a normal CRN, i.e., 1-step CRNs are just normal CRNs with initial configuration s_0. For the second step, we consider any configuration in TERM_1 combined with s_1 as a possible starting configuration and define $REACH_2$ to be the union of all reachable configurations

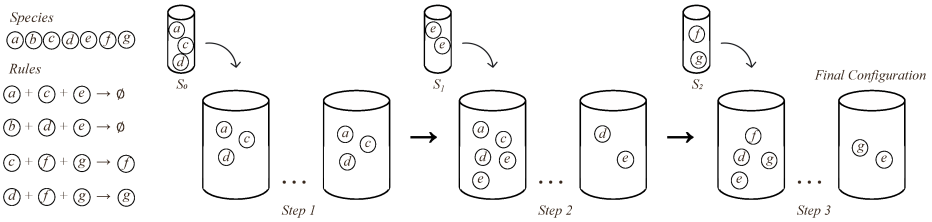


Fig. 1: An example step CRN system. The test tubes show the species added at each step and the system with those elements added. The CRN species and void rule-set are shown on the left.

from each possible starting configuration attained by adding s_1 to a configuration in $TERM_1$. We then define TERM_2 as the subset of configurations in REACH_2 that are terminal. Similarly, define $REACH_i$ to be the union of all reachable sets attained by using initial configuration 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 $I = s_0$.

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.

2.2 Computing Functions in Step CRNs

Here, we define what it means for a step CRN to compute a function $f(x_1, \cdots, x_n) = (y_1, \cdots, y_m)$ 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 string. 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 $x_1, \cdots, x_n$, the system results in a final terminal configuration whose output species encode the string $f(x_1, \cdots, x_n)$. For a fixed function f, the set of species s_i added at each step is fixed for all inputs to prevent encoding the output of the function within the configurations s_i.

Input-Strict Step CRN Computing. Given a Boolean function $f(x_1, \cdots, x_n) = (y_1, \cdots, y_m)$ 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 $S = ((\Lambda, \Gamma), (s_0, s_1, \cdots, s_{k-1}))$ is a step CRN, and $X = ((x_1^F, x_1^T), \cdots, (x_n^F, x_n^T))$ and $Y = ((y_1^F, y_1^T), \cdots, (y_m^F, y_m^T))$ are sequences of ordered-pairs with each $x_i^F, x_i^T, y_j^F, y_j^T \in \Lambda$. Given an n-input bit string $b = b_1, \cdots, b_n$, configuration $X(b)$ is defined as the configuration over Λ obtained by including one copy of x_i^F only if $b_i = 0$ and one copy of x_i^T only if $b_i = 1$ for each bit b_i. We consider this input to be strict, as opposed to allowing multiple copies of each input bit species. The corresponding step CRN $(\Lambda, \Gamma, (s_0 + X(b), \cdots, s_{k-1}))$ 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 $(\Lambda, \Gamma, (s_0 + X(b), \cdots, s_{k-1}))$, the terminal configuration contains at least 1 copy of y_j^F and 0 copies of y_j^T if the j^{th} bit of $f(b)$ is 0, and at least 1 copy of y_j^T and 0 copies of y_j^F 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. See [2] for a recent consideration of non-strict computation that utilizes bimolecular reactions. 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 [12] 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 [27].

2.3 Boolean and Threshold Circuits

A Boolean circuit on n variables $x_1, x_2, \cdots, x_n$ 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 $x_1, x_2, \cdots, x_n$, or labeled by constants 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 $x_1, x_2, \cdots, x_n$, 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).

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 TCs. 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 focuses on step CRNs consisting of $(3, 0)$ rules. Section 3.1 shows how to compute individual logic gates, and we give an example construction of a circuit in Section 3.2. We then present the general construction of computing TCs 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 1, 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 *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 assisted by species 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 $b \in \{T, F\}$. 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 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). 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's index and type i and g, respectively. 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 are 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 Fig. 2. Here, $|f_i^{in}| = 2$ and the initial configuration consists of the species x_1^T and x_2^F. By Table 2, this gate can be computed in five steps.

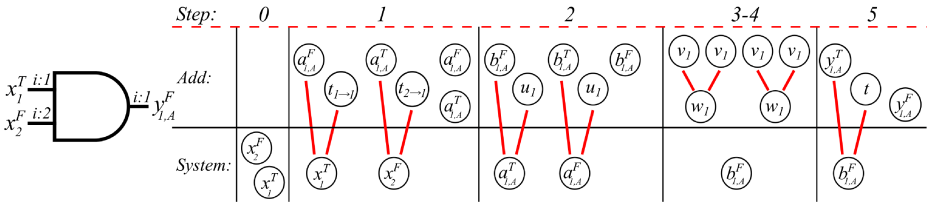


Fig. 2: Example AND gate and steps to compute using $(3, 0)$ rules. Note the gate indexing of the wires ($i:1$ and $i:2$) and the input indexing for the next level ($i:1$ since there is only one gate). The process of computing the gate is shown on the right in steps. The new species added at each step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

1. Two $a^T_{1,A}$, two $a^F_{1,A}$, one $t_{1\to1}$, and one $t_{2\to1}$ species are added to the system. This converts the two input species of the gate into $a^T_{1,A}$ and $a^F_{1,A}$ (causes all species except $a^T_{1,A}$ and $a^F_{1,A}$ to be deleted).
2. One $b^T_{1,A}$, two $b^F_{1,A}$, and two u_1 species are added. All species except a single $b^F_{1,A}$ 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 $b^F_{1,A}$ species remains.
5. One $y^T_{1,A}$, one $y^F_{1,A}$ and one t species are added. The $b^F_{1,A}$ species cause the $y^T_{1,A}$ and t species to be deleted. The $y^F_{1,A}$ 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 $a^b_{i,M}$ (step 1). The species retains the same index i and truth value b as the original input. If the fan-in of the majority gate is even, an extra *false* input species is added in. The species $b^b_{i,M}$ 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.

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.

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

Steps			Relevant Rules	Description
1	*Add*	$y_{i,N}^T$ $y_{i,N}^F$ $t_{j\to i}$	$y_{i,N}^T + x_j^T + t_{j\to i} \to \emptyset$ $y_{i,N}^F + x_j^F + t_{j\to i} \to \emptyset$	The output species ($y_{i,N}^b$) that is the complement of the input species (x_j^b) will be the only species remaining.

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

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_1^F).

3.3 Computing Circuits

For TC circuits with a max fan-out of 2, we show two methods of encoding the gates into the species that yield different results. The method in Lemma 1 reuses the gate species at each level of the circuit, and the method in Lemma 2 assigns unique species for each gate. Theorem 1 is generalized for TC circuits with arbitrary fan-out and combines the results from the Lemmas.

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

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

Lemma 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 $y_{n,B}^b$ species for each primary input with the appropriate indices and truth values. Section 3.1 explains how to compute TC gates. In order to apply a gate, we convert the outputs at an index i into the inputs for the next gate at index j. To simulate circuits with $O(W^2)$ species, we also must be able to reuse these input, output, and helper species. This is accomplished by having unique species for each gate at a given depth level. Figure 3a shows an example indexing.

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

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, $t_{2\to2}$	$y^T_{1,A} + x^T_1 + t_{1\to1} \to \emptyset$ $y^F_{2,A} + x^T_2 + t_{2\to2} \to \emptyset$
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**	$2b^T_{1,O}$, $2u_1$, $b^F_{1,O}$	$a^T_{1,O} + b^T_{1,O} + u_1 \to \emptyset$
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 the 'Steps' column shows the number and types of species being added at the beginning of that step.

Steps		**Relevant Rules**	**Description**
1	*Add* $\forall j \in f^{out}_i : x^T_j, x^F_j, t_{i\to j}$	$\forall j \in f^{out}_i$: $y^T_{i,g} + x^F_j + t_{i\to j} \to \emptyset$ $y^F_{i,g} + x^T_j + t_{i\to j} \to \emptyset$	$\forall j \in f^{out}_i$, convert $y^b_{i,g}$ output species into x^b_j input species.

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

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.

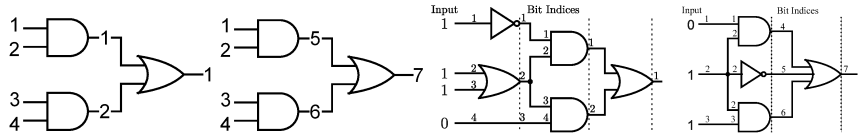
Lemma 2. *Threshold circuits (TC) with a max fan-out of 2 can be strictly computed by a step CRN with only* $(3, 0)$ *rules,* $O(G)$ *species,* $O(D)$ *steps, and* $O(W)$ *volume.*

Theorem 1. *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. This follows by expanding a given TC circuit to a fan-out 2 circuit and by applying the methods of Lemmas 1 and 2.

Steps		Relevant Rules	Description
1	*Add* $y^T_{i,g}, y^F_{i,g}, r$	$y^T_{i,g} + y^T_{i,g} + r \to \emptyset$ $y^F_{i,g} + y^F_{i,g} + r \to \emptyset$	Flip output's bit (e.g. if species $y^T_{i,g}$ is present, then delete it and preserve $y^F_{i,g}$)
2	*Add* $2y^T_{i,g}, 2y^F_{i,g}$	$y^T_{i,g} + y^T_{i,g} + y^T_{i,g} \to \emptyset$ $y^F_{i,g} + y^F_{i,g} + y^F_{i,g} \to \emptyset$	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.



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

Fig. 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.

4 TCs 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 as in Section 3.1. However, with catalysts, we 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^{in}_i be the set of all indices of the inputs fanning into a gate at index i. Let f^{out}_i 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^b_i or $y^b_{j\to i}$, 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^T_i) and a species representing a false output for each input ($\forall j \in f^{in}_i : y^F_{j\to i}$). To compute an OR gate instead, we add a species representing a true output ($y^T_{j\to i}$) for each input and a single y^F_i species. To compute NOT gates, we add one copy of each output species (y^b_i). For every

Gate Type	Step	Relevant Rules	Description
AND	$Add \begin{matrix} y_i^T \\ \forall j \in f_i^{in} : y_{j\to i}^F \end{matrix}$	$x_j^T + y_{j\to i}^F \to \emptyset$ $x_j^F + y_i^T \to \emptyset$	An input species with a certain truth value deletes the complement output species.
OR	$Add \begin{matrix} y_i^F \\ \forall j \in f_i^{in} : y_{j\to i}^T \end{matrix}$	$x_j^T + y_i^F \to \emptyset$ $x_j^F + y_{j\to i}^T \to \emptyset$	An input species with a certain truth value deletes the complement output species.
NOT	$Add \begin{matrix} y_i^T \\ y_i^F \end{matrix}$	$x_j^T + y_i^T \to \emptyset$ $x_j^F + y_i^F \to \emptyset$	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	$\begin{matrix} \lvert f_i^{in}\rvert \cdot a_i^T \\ \lvert f_i^{in}\rvert \cdot a_i^F \end{matrix}$	$\forall j \in f_i^{in}$: $x_j^T + a_i^F \to \emptyset$ $x_j^F + a_i^T \to \emptyset$	$\forall j \in f_i^{in}$, convert x_j^b input species into a_i^b species.
2	Add	$\begin{matrix} \lfloor \lvert f_i^{in}\rvert /2 \rfloor \cdot b_i^T \\ \lfloor \lvert f_i^{in}\rvert /2 \rfloor \cdot b_i^F \end{matrix}$	$a_i^T + b_i^F \to \emptyset$ $a_i^F + b_i^T \to \emptyset$	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	$\begin{matrix} y_i^T \\ y_i^F \end{matrix}$	$a_i^T + y_i^F \to \emptyset$ $a_i^F + y_i^T \to \emptyset$	Convert a_i^b into the proper output species (y_i^b).

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

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 rule $x_j^T + y_i^T \to \emptyset$ would be replaced by the rule $x_j^T + y_i^T \to x_j^T$.

OR Example. Consider an 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. Only the species $y_{2\to1}^T$ 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 retain the same truth value b as the original input and has gate index i. If the number of species fanning into the majority gate is even, an extra *false* input species is added. 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).

Initial Configuration: $y_1^F \quad y_2^T \quad y_3^T$					
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.*

4.2 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).

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

Theorem 2. *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.*
Due to space constraints, the proof is omitted.

4.4 Computing Circuits with (2,1) Catalyst Rules

Note that (2,1) catalyst rules are able to compute TCs alone. However, there is no known way to directly compute majority gates with (2,1) void rules, only (2,0). Thus, any majority gate is computed using AND, OR, and NOT gates when using only catalyst rules. Furthermore, deleting species that are no longer needed is slightly more convoluted with (2,1) rules compared to pure void rules.

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.*

Corollary 1. *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.*

Due to space constraints, the proof is omitted. The basic idea, however, is simply that it takes an additional log steps to handle the fan-in of the majority gates, which we can easily do with $(2, 1)$ catalyst rules.

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 $X \oplus Y$, i.e., 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 $X, Y_1, \cdots, Y_k$. 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 $c_0 - c_1$ (For two vectors $X = (x_1, \cdots, x_n)$ and $Y = (y_1, \cdots, y_n)$, $||X - Y||_1 = \sum_{i=1}^{n} |x_i - y_i|$).

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	$Add\ \begin{matrix} x_i^T \\ x_i^F \end{matrix}$	$y_i^T + x_i^F \to y_i^T$ $y_i^F + x_i^T \to y_i^F$ $\forall j \in f_i^{in}:$ $y_{j\to i}^T + x_i^F \to y_{j\to i}^T$ $y_{j\to i}^F + x_i^T \to y_{j\to i}^F$	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.*

Lemma 3. *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 at most $2r^s$.*

Due to space constraints, the proof is omitted. The configuration distance between two output configurations is related to the Hamming distance of the output strings they represent. Lemma 3 can be used to get a logarithmic lower bound for the number of steps required when we fix our rule size to be a constant.

Theorem 3. *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.*

Due to space constraints, the proof is omitted. 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 2 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 1 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 determining if a given (void) step CRN does compute a given function. Specifically, we consider the following decision problem, and show that with void rules it is coNP-hard (Theorem 4), and has coNP membership (Theorem 5). Due to space, the proofs are omitted.

Definition 4 ((Strict Function Verification)). *Given a step CRN $C_S = (S, X, Y)$ and a Boolean function $f(\cdot)$[4] 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 4. *It is coNP-hard to determine if a given $O(1)$-step CRN $C_S = (S, X, Y)$ with $(3, 0)$ rules computes the Boolean function $f_0(x_1, \cdots, x_n)$.*

Theorem 5. *Determining if a given 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.*

Theorem 6. *It is coNP-complete to determine if a given $O(1)$-step CRN $C_S = (S, X, Y)$ with $(3, 0)$ rules computes the Boolean function $f_0(x_1, \cdots, x_n)$.*

References

1. Alaniz, R.M., Fu, B., Gomez, T., Grizzell, E., Rodriguez, A., Schweller, R., Wylie, T.: Reachability in restricted chemical reaction networks (2022), arXiv:2211.12603
2. Anderson, R., Fu, B., Massie, A., Mukhopadhyay, G., Salinas, A., Schweller, R., Tomai, E., Wylie, T.: Computing threshold circuits with bimolecular void reactions in step chemical reaction networks. In: Proc. of the 21st Intl. Conf. on Unconventional Computation and Natural Computation. UCNC'24 (2024), to appear
3. Angluin, D., Aspnes, J., Diamadi, Z., Fischer, M.J., Peralta, R.: Computation in networks of passively mobile finite-state sensors. Distribed Computing **18**(4), 235–253 (mar 2006). https://doi.org/10.1007/s00446-005-0138-3
4. Angluin, D., Aspnes, J., Eisenstat, D.: A simple population protocol for fast robust approximate majority. Distributed Computing **21**, 87–102 (2008)
5. Angluin, D., Aspnes, J., Eisenstat, D., Ruppert, E.: The computational power of population protocols. Distributed Computing (2007)
6. Aris, R.: Prolegomena to the rational analysis of systems of chemical reactions. Archive for Rational Mechanics and Analysis **19**(2), 81–99 (jan 1965)
7. Aris, R.: Prolegomena to the rational analysis of systems of chemical reactions ii. some addenda. Arch. for Rational Mech. and Analysis **27**(5), 356–364 (jan 1968)
8. Arkin, A., Ross, J.: Computational functions in biochemical reaction networks. Biophysical journal **67**(2), 560–578 (1994)

[4] 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.

9. Aviram, A.: Molecules for memory, logic, and amplification. Journal of the American Chemical Society **110**(17), 5687–5692 (Aug 1988)
10. Cardelli, L., Kwiatkowska, M., Whitby, M.: Chemical reaction network designs for asynchronous logic circuits. Natural computing **17**, 109–130 (2018)
11. Cardelli, L., Tribastone, M., Tschaikowski, M.: From electric circuits to chemical networks. Natural Computing **19**, 237–248 (2020)
12. Chen, H.L., Doty, D., Soloveichik, D.: Deterministic function computation with chemical reaction networks. Natural computing **13**(4), 517–534 (2014)
13. Cook, M., Soloveichik, D., Winfree, E., Bruck, J.: Algorithmic Bioprocesses, chap. Programmability of Chemical Reaction Networks, pp. 543–584. Springer (2009)
14. Dalchau, N., Chandran, H., Gopalkrishnan, N., Phillips, A., Reif, J.: Probabilistic analysis of localized dna hybridization circuits. ACS synthetic biology **4**(8), 898–913 (2015)
15. Ellis, S.J., Klinge, T.H., Lathrop, J.I.: Robust chemical circuits. Biosystems **186**, 103983 (2019)
16. Fan, D., Wang, J., Han, J., Wang, E., Dong, S.: Engineering dna logic systems with non-canonical dna-nanostructures: Basic principles, recent developments and bio-applications. Science China Chemistry **65**(2), 284–297 (2022)
17. Hjelmfelt, A., Weinberger, E.D., Ross, J.: Chemical implementation of neural networks and turing machines. Proceedings of the National Academy of Sciences **88**(24), 10983–10987 (1991)
18. Jiang, H., Riedel, M.D., Parhi, K.K.: Digital logic with molecular reactions. In: Intl. Conf. on Computer-Aided Design. pp. 721–727. ICCAD'13, IEEE (2013)
19. Karp, R.M., Miller, R.E.: Parallel program schemata. Journal of Computer and System Sciences **3**(2), 147–195 (1969)
20. Mailloux, S., Guz, N., Zakharchenko, A., Minko, S., Katz, E.: Majority and minority gates realized in enzyme-biocatalyzed systems integrated with logic networks and interfaced with bioelectronic systems. Journal of Physical Chemistry B **118**(24), 6775–6784 (Jun 2014)
21. Petri, C.A.: Kommunikation mit Automaten. Ph.D. thesis, Rheinisch-Westfälischen Institutes für Instrumentelle Mathematik an der Universität Bonn (1962)
22. Qian, L., Winfree, E.: Scaling up digital circuit computation with dna strand displacement cascades. science **332**(6034), 1196–1201 (2011)
23. Qian, L., Winfree, E.: A simple dna gate motif for synthesizing large-scale circuits. Journal of The Royal Society Interface **8**(62), 1281–1297 (Feb 2011)
24. Sergeev, I.S.: Upper bounds for the formula size of symmetric boolean functions. Russian Mathematics **58**, 30–42 (2014)
25. Soloveichik, D., Cook, M., Winfree, E., Bruck, J.: Computation with finite stochastic chemical reaction networks. natural computing **7**(4), 615–633 (2008)
26. Soloveichik, D., Seelig, G., Winfree, E.: Dna as a universal substrate for chemical kinetics. Proc. of the National Academy of Sciences **107**(12), 5393–5398 (2010)
27. Thachuk, C., Condon, A.: Space and energy efficient computation with dna strand displacement systems. In: Intl. Workshop on DNA-Based Computers (2012)
28. Wang, B., Thachuk, C., Ellington, A.D., Winfree, E., Soloveichik, D.: Effective design principles for leakless strand displacement systems. Proceedings of the National Academy of Sciences **115**(52), E12182–E12191 (2018)
29. Winfree, E.: Chemical reaction networks and stochastic local search. In: DNA Computing and Molecular Programming. pp. 1–20. DNA'19, Springer (2019)
30. Xiao, W., Zhang, X., Zhang, Z., Chen, C., Shi, X.: Molecular full adder based on dna strand displacement. IEEE Access **8**, 189796–189801 (2020)