Probability 1 computation with chemical reaction networks*

Rachel Cummings† David Doty‡ David Soloveichik§

Abstract

The computational power of stochastic chemical reaction networks (CRNs) varies significantly with the output convention and whether or not error is permitted. Focusing on probability 1 computation, we demonstrate a striking difference between *stable* computation that converges to a state where the output cannot change, and the notion of *limit-stable* computation where the output eventually stops changing with probability 1. While stable computation is known to be restricted to semilinear predicates (essentially piecewise linear), we show that limit-stable computation encompasses the set of predicates $\phi : \mathbb{N} \to \{0, 1\}$ in Δ^0_2 in the arithmetical hierarchy (a superset of Turing-computable). In finite time, our construction achieves an error-correction scheme for Turing universal computation. We show an analogous characterization of the functions $f : \mathbb{N} \to \mathbb{N}$ computable by CRNs with probability 1, which encode their output into the count of a certain species. This work refines our understanding of the tradeoffs between error and computational power in CRNs.

1 Introduction

Recent advances in the engineering of complex artificial molecular systems have stimulated new interest in models of chemical computation. How can chemical reactions process information, make decisions, and solve problems? A natural model for describing abstract chemical systems in a well-mixed solution is that of (finite) chemical reaction networks (CRNs), i.e., finite sets of chemical reactions such as $A + B \to A + C$. Subject to discrete semantics (integer number of molecules) the model corresponds to a continuous time, discrete state, Markov process. A state of the system is a vector of non-negative integers specifying the molecular counts of the species (e.g., A, B, C), a reaction can occur only when its reactants are present, and transitions between states correspond to reactions (i.e., when the above reaction occurs the count of B is decreased by 1 and the count of C increased by 1). CRNs are used extensively to describe natural biochemical systems in the cellular context. Other natural sciences use the model as well: for example in ecology an equivalent model is widely employed to describe population dynamics [18]. Recently, CRNs began serving as a programming language for engineering artificial chemical systems. In particular, DNA strand displacement systems have the flexibility to realize arbitrarily complex interaction rules [4, 6, 17], demonstrating that arbitrary CRNs have a chemical implementation. Outside of chemistry, engineering of sensor networks and robot swarms often uses CRN-like specification rules to prescribe behavior [2].

*The first author was supported by NSF grants CCF-1049899 and CCF-1217770, the second author was supported by NSF grants CCF-1219274 and CCF-1162589 and the Molecular Programming Project under NSF grant 1317694, and the third author was supported by NIGMS Systems Biology Center grant P50 GM081879.

†Northwestern University, Evanston, IL, USA, rachelc@u.northwestern.edu

‡California Institute of Technology, Pasadena, CA, USA, ddoty@caltech.edu

§University of California, San Francisco, San Francisco, CA, USA, david.soloveichik@ucsf.edu

The exploration of the computational power of CRNs rests on a strong theoretical foundation, with extensive connections to other areas of computing. Similar models have arisen repeatedly in theoretical computer science: Petri nets [12], vector addition systems [10], population protocols [2], etc. They share the fundamental feature of severely limited "agents" (molecules), with complex computation arising only through repeated interactions between multiple agents. In population protocols it is usually assumed that the population size is constant, while in CRNs molecules could be created or destroyed[1] — and thus different questions are sometimes natural in the two settings.

Informally speaking, we can identify two general kinds of computation in CRNs. In *non-uniform* computation, a single CRN computes a function over a finite domain. This is analogous to Boolean circuits in the sense that any given circuit computes only on inputs of a particular size (number of bits), and to compute on larger inputs a different circuit is needed. Conversely, in *uniform* computation, a single CRN computes on all possible input vectors. This is analogous to Turing machines that are expected to handle inputs of arbitrary size placed on their (unbounded) input tape. In this work we focus entirely on uniform computation.

Previous research on uniform computation has emphasized the difference in computational power between paradigms intended to capture the intuitive notions of error-free and error-prone computation [7]. In contrast to many other models of computing, a large difference was identified between the two settings for CRNs. We now review the previously studied probability 1 and probability < 1 settings. In the main result of this paper we develop an error correction scheme that reduces the error of the output with time and achieves probability 1 computation in the limit. Thus, the large distinction between probability 1 and probability < 1 computation surprisingly disappears in a "limit computing" setting.

The best studied type of probability 1 computation incorporates the stable output criterion (Table 1, prob correct = 1, stable) and was shown to be limited to semilinear predicates [1] (later extended to functions [5]). For example, consider the following CRN computing the parity predicate (a semilinear predicate). The input is encoded in the initial number of molecules of X, and the output is indicated by species Y (yes) and N (no):

$$X + Y \rightarrow N$$
$$X + N \rightarrow Y$$

Starting with the input count $n \in \mathbb{N}$ of X, as well as $1Y$, the CRN converges to a state where a molecule of the correct output species is present (Y if n is even and N if n is odd) and the incorrect output species is absent. From that point on, no reaction can change the output.

[1] A CRN where reactions can increase or decrease the count of molecules may not be consistent with the conservation of mass for a closed system. In that case the CRN represents the behavior of an open system with an implicit, potentially unbounded (or replenishable) reservoir.

	committing	stable	limit-stable
prob correct $= 1$	(constant)	semilinear [1]	Δ_2^0 [this work]
prob correct < 1	computable [16]	(computable)	(Δ_2^0)

Table 1: Categorizing the computational power of CRNs by output convention and allowed error probability. In all cases we consider the class of total predicates $\phi : \mathbb{N}^k \to \{\text{no}, \text{yes}\}$ ("total" means ϕ must be defined on all input values). The input consists of the initial molecular counts of input species $X_1, \ldots, X_k$. *Committing*: In this output convention, producing any molecules of N indicates that the output of the whole computation is "no", and producing any molecules of Y indicates "yes". *Stable*: Let the *output of a state* be "no" if there are some N molecules and no Y molecules (and vice versa for "yes"). In the stable output convention, the output of the whole computation is $b \in \{\text{no}, \text{yes}\}$ when the CRN reaches a state with output value b from which every reachable state also has output value b. *Limit-stable*: The output of the computation is considered $b \in \{\text{no}, \text{yes}\}$ when the CRN reaches a state with output b and never changes it again (even though states with different output may remain reachable). The parenthetical settings have not been explicitly formalized; however the computational power indicated naturally follows by extending the results from other settings.[2]

Motivated by such examples, probability 1 *stable* computation admits the changing of output as long as the system converges with probability 1 to an *output stable state* — a state from which no sequence of reactions can change the output. In the above example, the states with X absent are output stable. (Although the limits on stable computation were proven in the population protocols model [1] they also apply to general CRNs where infinitely many states may be reachable.)

A more stringent output convention requires irreversibly producing N or Y to indicate the output (i.e., "Y is producible from initial state $\mathbf{x}$" $\iff$ "N is not producible from $\mathbf{x}$" $\iff$ $\phi(\mathbf{x}) = \text{yes}$). We term such output convention *committing*. However, probability 1 committing computation is restricted to constant predicates (see Table 1, prob correct = 1, committing, and the footnote therein).

Intuitively, committing computation "knows" when it is done, while stable computation does not. Nonetheless, stable computation is not necessarily impractical. All semilinear predicates can be stably computed such that checking for output stability is equivalent to simply inspecting whether any further reaction is possible — so an outside observer can easily recognize the completion of computation [3]. While stable CRNs do not know when they are done computing, different downstream processes can be catalyzed by the N and Y species. As long as these processes can be

[2]Probability 1 committing computation: Suppose there are two inputs $\mathbf{x}_1, \mathbf{x}_2 \in \mathbb{N}^k$ such that $\phi(\mathbf{x}_1) = \text{no}$ and $\phi(\mathbf{x}_2) = \text{yes}$. Consider any input $\mathbf{x}_3$ such that $\mathbf{x}_3 \geq \mathbf{x}_1$ and $\mathbf{x}_3 \geq \mathbf{x}_2$. From state $\mathbf{x}_3$ we can produce N by the sequence of reactions from $\phi(\mathbf{x}_1)$, and Y by the sequence of reactions from $\phi(\mathbf{x}_2)$. Both sequences occur with some non-zero probability, and so the probability of error is non-zero for non-constant predicates. Probability < 1 stable computation: To show that this output convention admits no more than computable predicates, note that the question of whether there is a sequence of reactions to change the output is computable. Thus, if we are guaranteed that with probability larger than $1/2$ we reach the correct output stable state, then by computing the probabilities of the (infinite) Markov chain of all reachable states far enough, we are guaranteed to identify the correct output. Probability < 1 limit-stable computation: Our negative result for probability 1 limit-stable computation can be modified to apply.

Note that particulars of input encoding generally make a difference for the weaker computational settings. For instance, probability 1 committing computation can compute more than constant predicates if they are not required to be total (e.g. a committing CRN can certainly distinguish between two $\leq$-incomparable inputs). Indeed, from a non-uniform perspective, this setting is in some sense maximally powerful since arbitrary Boolean circuits can be computed (with a "dual-rail" encoding of signals).

undone by the presence of the opposite species, the overall computation can be in this sense stable as well. A canonical downstream process is signal amplification in which a much larger population of $\hat{N}, \hat{Y}$ is interconverted by reactions $\hat{N} + Y \to \hat{Y} + Y$ and $\hat{Y} + N \to \hat{N} + N$. Finally all semilinear predicates can be stably computed quickly (polylogarithmic in the input molecular counts).

In contrast to the limited computational abilities of the probability 1 settings just now discussed, tolerating a positive probability of error significantly expands computational power. Indeed arbitrary computable functions (Turing universal computation) can be computed with the committing output convention [16]. Turing universal computation can also be fast — the CRN simulation incurs only a polynomial slowdown. However, error is unavoidable and is due fundamentally to the inability of CRNs to deterministically detect the absence of species. (Note that Turing universal computation is only possible in CRNs when the reachable state space, i.e., molecular count, is unbounded — and is thus not meaningful in population protocols.)

When a CRN is simulating a Turing machine, errors in simulation cannot be avoided. However, can they be *corrected* later? In this work we develop an error correction scheme that can be applied to Turing universal computation that ensures overall output error decreases the longer the CRN runs. Indeed, in the limit of time going to infinity, with probability 1 the answer is correct.

To capture Turing-universal probability 1 computation with such an error correction process, a new output convention is needed, since the committing and stable conventions are limited to much weaker forms of probability 1 computation (Table 1). *Limit-stability* subtly relaxes the stability requirement: instead of there being no path to change the output, we require the system to eventually stop taking such paths with probability 1. To illustrate the difference between the original notion of stability and our notion of *limit-stability*, consider the reachability of the empty state (without any molecules of Y, in which the output is undefined) in the CRN:

$$\varnothing \xrightarrow{1} Y$$

$$Y \xrightarrow{1} \varnothing$$

$$Y \xrightarrow{2} 2Y$$

From any reachable state, the empty state is reachable (just execute the second reaction enough times). However, with probability 1, the empty state is visited only a finite number of times.[3] If, as before, we think of the presence of Y indicating a yes output, then the yes output is never stable (the empty state is always reachable), but with probability 1 a yes output will be produced and never change — i.e., it is limit-stable. Thus the above CRN computes the constant predicate $\phi = 1$ under the limit-stable convention but not the stable convention.[4]

We show that with the limit-stable output convention, errors in Turing universal computation can be rectified eventually with probability 1. Our construction is based on simulating a register machine (a.k.a. Minsky counter machine) over and over in an infinite loop, increasing the number of

[3] The second and third reactions are equivalent to the gambler's ruin problem [8], which tells us that, because the probability of increasing the count of Y is twice that of decreasing its count, there is a positive probability that Y's count never reaches 0. The first reaction can only increase this probability. Since whenever Y's count reaches 0, we have another try, eventually with probability 1 we will not stop visiting the state $\{0Y\}$.

[4] The notions of stable and limit-stable are incomparable in the sense that a CRN can compute a predicate under one convention but not the other. The previous paragraph showed a CRN that limit-stably computes a predicate but does not stably compute it. To see the other direction, consider removing the first reaction and starting in state $\{1N, 1Y\}$, i.e., a no voter and a yes voter. This state has undefined output, as does any state with positive count of Y, since there is always an N present. The state $\{1N, 0Y\}$ is has defined output "no" and is stable (since we removed the first reaction). Since that state is always reachable, the CRN stably computes the predicate $\phi = 0$. However, there is a positive probability that $\{1N, 0Y\}$ is never reached, so the predicate is not computed under the limit-stable convention.

simulated steps each time (dovetailing). Each time the CRN updates its answer to the answer given by the most recently terminated simulation. While errors may occur during these simulations, our construction is designed such that with probability 1 only a finite number of errors occur (by the Borel-Cantelli Lemma), and then after some point the output will stay correct forever. The main difficulty is ensuring that errors "fail gracefully": they are allowed to cause the wrong answer to appear for a finite time, but they cannot, for instance, interfere with the dovetailing itself. We also show that the expected time to stabilize to the correct output is polynomial in the running time of the register machine (which, however, is exponentially slower than a Turing machine). This implies that an outside observer can read out the output after time that is polynomial in the number of steps of the register machine and be guaranteed the correct output with high probability (as in the probability < 1 setting), but waiting longer eventually pushes that probability arbitrarily close to 1.

In computability theory, if a Turing machine is allowed to change its output a finite (but unknown) number of times, then it can decide more than the Turing-computable predicates. Rather, Turing machines "limit computing" in this manner can decide the larger class of predicates known as Δ^0_2 (the second level of the arithmetical hierarchy [13, 15]). Since in limit-stable probability 1 computation, the CRN can change its output, it is natural to wonder whether the class of predicates computed also corresponds to Δ^0_2. Indeed, we show that the class of predicates limit-stable computable by CRNs with probability 1 is exactly the Δ^0_2 predicates. Note that since we do not know when the CRN will stop changing its answer[5], the output to which a limit-stable computation converges cannot be practically read out in finite time (analogously to limit computation with Turing machines).

Relaxing the definition of probability 1 computation further does not increase computational power. An alternative definition of probability 1 limit-stable computation is to require that as time goes to infinity, the probability of expressing the correct output approaches 1. In contrast to limit-stability, the output may change infinitely often, so long as the frequency of time the output is incorrect approaches 0. Note that this is exactly the distinction between almost sure convergence and convergence in probability. Our proof that probability 1 limit-stable computation is limited to Δ^0_2 predicates applies to this weaker sense of convergence as well, bolstering the generality of our result. Interestingly, it is still unclear whether in CRNs there is a natural definition of probability 1 computation that exactly corresponds to the class of Turing-computable functions.

In the remainder of this paper we focus on the type of probability 1 computation captured by the notion of limit-stability, reserving the term *probability* 1 *computation* to refer specifically to probability 1 limit-stable computation.

To our knowledge, the first hints in the CRN literature of probability 1 Turing universal computation occur in ref. [19], where Zavattaro and Cardelli showed that the following question is uncomputable: Will a given CRN with probability 1 reach a state where no further reactions are possible? Although their construction relied on repeated simulations of a Turing machine, it did not use the Borel-Cantelli Lemma, and could not be directly applied to computation with output.

2 Preliminaries

Our main technique uses the following classical result, known as the Borel-Cantelli Lemma:

[5]Consider the contrast to stable CRN computation. Although stably computing CRNs do not know when they are finished, an outside observer *can* compute if the CRN is done – i.e., no sequence of reactions can change the output, since this is easily reduced to the problems of deciding whether one state is reachable from another, known to be decidable [11], and deciding if a superset of a state is reachable from another, also decidable [10].

Lemma 2.1 (Borel-Cantelli Lemma)**.** *Let $E_0, E_1, \ldots$ be an infinite sequence of events in some probability space, such that* $\sum\limits_{i=0}^{\infty} \Pr[E_i] < \infty$*. Then* $\Pr[$*finitely many E_i occur*$] = 1$.

2.1 Computability theory

We use the term *predicate* (a.k.a. *language*, *decision problem*) interchangeably to mean a subset $L \subseteq \mathbb{N}^k$, or equivalently a function $\phi : \mathbb{N}^k \to \{0, 1\}$, such that $\phi(\mathbf{x}) = 1 \iff \mathbf{x} \in L$.

We say that a predicate $\phi : \mathbb{N}^k \to \{0, 1\}$ is *limit computable* if there is a computable function $r : \mathbb{N}^k \times \mathbb{N} \to \{0, 1\}$ such that, for all $\mathbf{x} \in \mathbb{N}^k$, $\lim\limits_{t\to\infty} r(\mathbf{x}, t) = \phi(\mathbf{x})$. The following equivalence is known as the Shoenfield limit lemma [15]; it is not required to understand our main theorem or proofs, but it provides an alternative characterization of the limit computable predicates that gives evidence that the definition is "natural."

Lemma 2.2 ([14])**.** *A predicate ϕ is limit computable if and only if it is Turing reducible to the halting problem.*

We write Δ^0_2 to denote the class of all limit computable predicates.

2.2 Chemical reaction networks

If Λ is a finite set (in this paper, of chemical species), we write $\mathbb{N}^\Lambda$ to denote the set of functions $f : \Lambda \to \mathbb{N}$. Equivalently, assuming some canonical ordering of the elements of Λ, we view an element $\mathbf{c} \in \mathbb{N}^\Lambda$ as a vector $\mathbf{c} \in \mathbb{N}^{|\Lambda|}$ of $|\Lambda|$ nonnegative integers, with each coordinate "labeled" by an element of Λ. Given $S \in \Lambda$ and $\mathbf{c} \in \mathbb{N}^\Lambda$, we refer to $\mathbf{c}(S)$ as the *count of S in* $\mathbf{c}$. We write $\mathbf{c} \leq \mathbf{c}'$ if $\mathbf{c}(S) \leq \mathbf{c}'(S)$ for all $S \in \Lambda$, and $\mathbf{c} < \mathbf{c}'$ if $\mathbf{c} \leq \mathbf{c}'$ and $\mathbf{c} \neq \mathbf{c}'$. Given $\mathbf{c}, \mathbf{c}' \in \mathbb{N}^\Lambda$, we define the vector component-wise operations of addition $\mathbf{c} + \mathbf{c}'$ and subtraction $\mathbf{c} - \mathbf{c}'$. For a set $\Delta \subset \Lambda$, we view a vector $\mathbf{c} \in \mathbb{N}^\Delta$ equivalently as a vector $\mathbf{c} \in \mathbb{N}^\Lambda$ by assuming $\mathbf{c}(S) = 0$ for all $S \in \Lambda \setminus \Delta$.

Given a finite set of chemical species Λ, a *reaction* over Λ is a triple $\alpha = \langle \mathbf{r}, \mathbf{p}, k\rangle \in \mathbb{N}^\Lambda \times \mathbb{N}^\Lambda \times \mathbb{R}^+$, specifying the stoichiometry (amount consumed/produced) of the reactants and products, respectively, and the *rate constant* k. For instance, given $\Lambda = \{A, B, C\}$, the reaction $A+2B \overset{7.5}{\to} A+3C$ is represented by the triple $\langle(1, 2, 0),(1, 0, 3), 7.5\rangle$. If not specified, assume that the rate constant $k = 1$. A *chemical reaction network (CRN)* is a pair $N = (\Lambda, R)$, where Λ is a finite set of chemical *species*, and R is a finite set of reactions over Λ. A *state* of a CRN $N = (\Lambda, R)$ is a vector $\mathbf{c} \in \mathbb{N}^\Lambda$.

Given a state $\mathbf{c}$ and reaction $\alpha = \langle \mathbf{r}, \mathbf{p}\rangle$, we say that α is *applicable* to $\mathbf{c}$ if $\mathbf{r} \leq \mathbf{c}$ (i.e., $\mathbf{c}$ contains enough of each of the reactants for the reaction to occur). If α is applicable to $\mathbf{c}$, then write $\alpha(\mathbf{c})$ to denote the state $\mathbf{c}+\mathbf{p}-\mathbf{r}$ (i.e., the state that results from applying reaction α to $\mathbf{c}$). If $\mathbf{c}' = \alpha(\mathbf{c})$ for some reaction $\alpha \in R$, we write $\mathbf{c} \to^1 \mathbf{c}'$. An *execution sequence* $\mathcal{E}$ is a finite or infinite sequence of states $\mathcal{E} = (\mathbf{c}_0, \mathbf{c}_1, \mathbf{c}_2, \ldots)$ such that, for all $i \in \{1, \ldots, |\mathcal{E}| - 1\}$, $\mathbf{c}_{i-1} \to^1 \mathbf{c}_i$. If a finite execution sequence starts with $\mathbf{c}$ and ends with $\mathbf{c}'$, we write $\mathbf{c} \to \mathbf{c}'$, and we say that $\mathbf{c}'$ is *reachable from* $\mathbf{c}$.

2.3 Stable decidability by CRNs

We now review the definition of stable decidability of predicates introduced by Angluin, Aspnes, and Eisenstat [1]. Although our main theorem concerns probability 1 decidability, not stable decidability, many of the definitions of this section will be required, so it it useful to review. Intuitively, some species "vote" for a yes/no answer, and a CRN is a stable decider if it is guaranteed to reach a consensus vote that cannot change.

A *chemical reaction decider* (CRD) is a tuple $\mathcal{D} = (\Lambda, R, \Sigma, \Upsilon, \phi, \mathbf{s})$, where (Λ, R) is a CRN, $\Sigma \subseteq \Lambda$ is the *set of input species*,[6] $\Upsilon \subseteq \Lambda$ is the set of *voters*, $\phi : \Upsilon \to \{0, 1\}$ is the *(Boolean) output function*, and $\mathbf{s} \in \mathbb{N}^{\Lambda \setminus \Sigma}$ is the *initial context*. An input to $\mathcal{D}$ is a vector $\mathbf{x} \in \mathbb{N}^{\Sigma}$, or equivalently $\mathbf{x} \in \mathbb{N}^k$ if $|\Sigma| = k$; $\mathcal{D}$ and $\mathbf{x}$ define an *initial state* $\mathbf{i} \in \mathbb{N}^{\Lambda}$ as $\mathbf{i} = \mathbf{s} + \mathbf{x}$ (when $\mathbf{i}$ and $\mathbf{x}$ are considered as elements of $\mathbb{N}^{\Lambda}$).[7] When $\mathbf{i}$ is clear from context, we say that a state $\mathbf{c}$ is *reachable* if $\mathbf{i} \to \mathbf{c}$.

We extend ϕ to a partial function $\Phi : \mathbb{N}^{\Lambda} \dashrightarrow \{0, 1\}$ as follows. $\Phi(\mathbf{c})$ is undefined if either $\mathbf{c}(V) = 0$ for all $V \in \Upsilon$, or if there exist $N, Y \in \Upsilon$ such that $\mathbf{c}(N) > 0$, $\mathbf{c}(Y) > 0$, $\phi(N) = 0$ and $\phi(Y) = 1$. Otherwise, $(\exists b \in \{0, 1\})(\forall V \in \Upsilon)(\mathbf{c}(V) > 0 \implies \phi(V) = b)$; in this case, the *output* $\Phi(\mathbf{c})$ of state $\mathbf{c}$ is b. In other words $\Phi(\mathbf{c}) = b$ if some voters are present and they all vote b.

If $\Phi(\mathbf{y})$ is defined and every state $\mathbf{y}'$ reachable from $\mathbf{y}$ satisfies $\Phi(\mathbf{y}) = \Phi(\mathbf{y}')$, then we say that $\mathbf{y}$ is *output stable*, i.e., if $\mathbf{y}$ is ever reached, then no sequence of reactions can ever change the output. We say that $\mathcal{D}$ *stably decides* the predicate $\phi : \mathbb{N}^k \to \{0, 1\}$ if, for all input states $\mathbf{x} \in \mathbb{N}^k$, for every state $\mathbf{c}$ reachable from $\mathbf{x}$, there is an output stable state $\mathbf{y}$ reachable from $\mathbf{c}$ such that $\Phi(\mathbf{y}) = \phi(\mathbf{x})$. In other words, no sequence of reactions (reaching state $\mathbf{c}$) can *prevent* the CRN from being able to reach the correct answer (since $\mathbf{c} \to \mathbf{y}$ and $\Phi(\mathbf{y}) = \phi(\mathbf{x})$) and staying there if reached (since $\mathbf{y}$ is output stable).

At first glance, this definition appears too weak to claim that the CRN is "guaranteed" to reach the correct answer. It merely states that the CRN is guaranteed stay in states from which it is *possible* to reach the correct answer. If the set of states reachable from the initial state $\mathbf{x}$ is finite (as with all population protocols since the total molecular count is bounded), however, then it is easy to show that with probability 1, the CRN will eventually reach an output stable state with the correct answer, assuming the system's evolution is governed by stochastic chemical kinetics, defined later.

Why is this true? By the definition of stable decidability, for all reachable states $\mathbf{c}$, there exists a reaction sequence $r_{\mathbf{c}}$ such that some output stable state $\mathbf{y}$ is reached after applying $r_{\mathbf{c}}$ to $\mathbf{c}$. If there are a finite number of reachable states, then there exists $\epsilon > 0$ such that for all reachable states $\mathbf{c}$, $\Pr[r_{\mathbf{c}}$ occurs upon reaching $\mathbf{c}] \geq \epsilon$. For any such state $\mathbf{c}$ visited ℓ times, the probability that $r_{\mathbf{c}}$ is *not* followed after all ℓ visits is at most $(1 - \epsilon)^{\ell}$, which approaches 0 as $\ell \to \infty$. Since some state $\mathbf{c}$ must be visited infinitely often in a reaction sequence that avoids an output stable state $\mathbf{y}$ forever, this implies that the probability is 0 that $r_{\mathbf{c}}$ is never followed after reaching $\mathbf{c}$, showing that "stable computation" implies "probability 1 computation" if the reachable state space is finite. To see the converse, suppose that a reachable state $\mathbf{c}$ exists from which no correct output-stable state is reachable. Since $\mathbf{c}$ is reached with positive probability, the CRN has probability strictly less than 1 to reach a correct output stable state. Thus, with a finite reachable state space, "stable computation" and "probability 1 computation" are equivalent.

[6] In Section 3 and beyond, we restrict attention to the case that $|\Sigma| = 1$, i.e., single-integer inputs. Since our main result will show that the predicates computable with probability 1 by a CRN encompass all of Δ_2^0, this restriction will not be crucial, since any computable encoding function $e : \mathbb{N}^k \to \mathbb{N}$ that represents k-tuples of integers as a single integer, and its inverse decoding function $d : \mathbb{N} \to \mathbb{N}^k$, can be computed by the CRN to handle a k-tuples of inputs that is encoded into the count of a single input species X. Such encoding functions are provably not computable by semilinear functions, so this distinction is more crucial in the realm of stable computation by CRDs, which are limited to semilinear predicates and functions [1, 5].

[7] In other words, species in $\Lambda \setminus \Sigma$ must always start with the same counts, and counts of species in Σ are varied to represent different inputs to $\mathcal{D}$, similarly to a Turing machine that starts with different binary string inputs, but the Turing machine must always start with the same initial state and tape head position.

2.4 Probability 1 decidability by CRNs

In order to define probability 1 computation with CRNs, we first review the model of stochastic chemical kinetics. It is widely used in quantitative biology and other fields dealing with chemical reactions between species present in small counts [9]. It ascribes probabilities to execution sequences, and also defines the time of reactions.

A reaction is *unimolecular* if it has one reactant and *bimolecular* if it has two reactants. We use no higher-order reactions in this paper.

The kinetics of a CRN is described by a continuous-time Markov process as follows. The system has some volume $v \in \mathbb{R}^+$ that affects the transition rates, which may be fixed or allowed to vary over time; in this paper we assume a constant volume of 1.[8] In state $\mathbf{c}$, the *propensity* of a unimolecular reaction $\alpha : X \xrightarrow{k} \ldots$ in state $\mathbf{c}$ is $\rho(\mathbf{c}, \alpha) = k \cdot \mathbf{c}(X)$. The propensity of a bimolecular reaction $\alpha : X + Y \xrightarrow{k} \ldots$, where $X \neq Y$, is $\rho(\mathbf{c}, \alpha) = k \cdot \frac{\mathbf{c}(X) \cdot \mathbf{c}(Y)}{v}$. The propensity of a bimolecular reaction $\alpha : X + X \xrightarrow{k} \ldots$ is $\rho(\mathbf{c}, \alpha) = \frac{k}{2} \cdot \frac{\mathbf{c}(X) \cdot (\mathbf{c}(X) - 1)}{v}$. The propensity function governs the evolution of the system as follows. The time until the next reaction occurs is an exponential random variable with rate $\rho(\mathbf{c}) = \sum_{\alpha \in R} \rho(\mathbf{c}, \alpha)$ (note $\rho(\mathbf{c}) = 0$ if and only if no reactions are applicable to $\mathbf{c}$). The probability that next reaction will be a particular α_{next} is $\frac{\rho(\mathbf{c}, \alpha_{\text{next}})}{\rho(\mathbf{c})}$.

We now define probability 1 computation by CRDs. Our definition is based on the *limit-stable* output convention as discussed in the introduction. Let $\mathcal{D} = (\Lambda, R, \Sigma, \Upsilon, \phi, \mathbf{s})$ be a CRD. Let $\mathcal{E} = (\mathbf{c}_0, \mathbf{c}_1, \ldots)$ be an execution sequence of $\mathcal{D}$. In general $\mathcal{E}$ could be finite or infinite, depending on whether $\mathcal{D}$ can reach a terminal state (one in which no reaction is applicable); however, in this paper all CRDs will have no reachable terminal states, so assume $\mathcal{E}$ is infinite. We say that $\mathcal{E}$ has a defined output if there exists $b \in \{0, 1\}$ such that, for all but finitely many $i \in \mathbb{N}$, $\Phi(\mathbf{c}_i) = b$.[9] In other words, $\mathcal{E}$ eventually stabilizes to a certain answer. In this case, write $\Phi(\mathcal{E}) = b$; otherwise, let $\Phi(\mathcal{E})$ be undefined.

If $\mathbf{x} \in \mathbb{N}^k$, write $\mathcal{E}(\mathcal{D}, \mathbf{x})$ to denote the random variable representing an execution sequence of $\mathcal{D}$ on input $\mathbf{x}$, resulting from the Markov process described previously. We say that $\mathcal{D}$ *decides* $\phi : \mathbb{N}^k \to \{0, 1\}$ *with probability 1* if, for all $\mathbf{x} \in \mathbb{N}^k$, $\Pr[\Phi(\mathcal{E}(\mathcal{D}, \mathbf{x})) = \phi(\mathbf{x})] = 1$.

3 Turing-decidable predicates

This section describes how a CRD can decide an arbitrary Turing-decidable predicate with probability 1. This construction also contains most of the technical details needed to prove our positive result that CRDs can decide arbitrary Δ_2^0 predicates with probability 1. The proof is via simulation of register machines, which are able to simulate arbitrary Turing machines if at least 3 registers are used. This section describes the simulation and gives intuition for how it works. Section 4 proves its correctness. Section 5 shows how to extend the construction to handle Δ_2^0 predicates and prove that no more predicates can be decided with probability 1 by a CRD.

[8] A common restriction is to assume the *finite density constraint*, which stipulates that arbitrarily large mass cannot occupy a fixed volume, and thus the volume must grow proportionally with the total molecular count. With some minor modifications to ensure *relative* rates of reactions stay the same (even though all bimolecular reactions would be slowed down in absolute terms), our construction would work under this assumption, although the time analysis would change. For the sake of conceptual clarity, we present the construction assuming a constant volume. The issue is discussed in more detail in Section 4.

[9] Note that this is equivalent to requiring $\lim_{i \to \infty} \Phi(\mathbf{c}_i) = b$, hence the term "limit" in the phrase "limit-stable" comes from this requirement on infinite executions with a well-defined limit output.

3.1 Register machines

A *register machine* M has m *registers* $r_1, \ldots, r_m$ that can each hold a non-negative integer. M is programmed by a finite sequence (*lines*) of *instructions*. There are four types of instructions: `accept`, `reject`, `inc(`r`)`, and `dec(`r`,`k`)`. For simplicity, we describe our construction for single-input register machines and thus predicates $\phi : \mathbb{N} \to \{0, 1\}$, but it can be easily extended to more inputs. The input $n \in \mathbb{N}$ to M is the initial value of register r_1, and the remaining $m - 1$ registers are used to perform a computation on the input, which by convention are set to 0 initially. The semantics of execution of M is as follows. The *initial line* is the first instruction in the sequence. If the current line is `accept` or `reject`, then M halts and accepts or rejects, respectively. If the current line is `inc(`r`)`, then register r is incremented, and the next line in the sequence becomes the current line. If the current line is `dec(`r`,`k`)`, then register r is decremented, and the next line in the sequence becomes the current line, unless $r = 0$, in which case it is left at 0 and line k becomes the current line. In other words, M executes a straight-line program, with a "conditional jump" that occurs when attempting to decrement a 0-valued register. For convenience we assume there is a fifth type of instruction `goto(`k`)`, meaning "unconditionally jump to line k". This can be indirectly implemented by decrementing a special register r_0 that always has value 0, or easily implemented in a CRN directly. Assuming the machine halts on all inputs, the set of input values n that cause the machine to accept is then the language/predicate decided by the machine. For example, the following register machine decides the parity of the initial value of register r_1:

1: `dec(`r_1`,5)`
2: `dec(`r_1`,4)`
3: `goto(1)`
4: `accept`
5: `reject`

Chemical reaction networks can be used to simulate any register machine through a simple yet error-prone construction, which is similar to the simulation described in [16]. We now describe the simulation and highlight the source of error. Although this simulation may be error-prone, the effect of the errors has a special structure, and our main construction will take advantage of this structure to keep errors from invalidating the entire computation. Specifically, there is a possibility of an error precisely when the register machine performs a conditional jump.

For a register machine with l lines of instructions and m registers, create molecular species $L_1, \ldots, L_l$ and $R_1, \ldots, R_m$, and to represent input n, the initial configuration is $\{nR_1, 1L_1\}$. The presence of molecule L_i is used to indicate that the current line is i. Since the register machine can only be in one line at a time, there will be exactly one molecule of the form L_i present in the solution at any time. The count of species R_j represents the current value of register r_j. The following table shows the reactions to simulate an instruction of the register machine, assuming the instruction occurs on line i:

`accept`	$L_i \to H_Y$
`reject`	$L_i \to H_N$
`goto(`k`)`	$L_i \to L_k$
`inc(`r_j`)`	$L_i \to L_{i+1} + R_j$
`dec(`r_j`,`k`)`	$L_i + R_j \to L_{i+1}$ $L_i \to L_k$

The first four reactions are error-free simulations of the corresponding instructions. The final two reactions are an error-prone way to decrement register r_j. If $r_j = 0$, then only the latter reaction is possible, and when it occurs it is a correct simulation of the instruction. However, if $r_j > 0$ (hence there are a positive number R_j molecules in solution), then either reaction is possible. While only the former reaction is correct, the latter reaction could still occur. The semantic effect this has on the register machine being simulated is that, when a decrement $\texttt{dec}(r_j,k)$ is possible because $r_j > 0$, the machine may nondeterministically jump to line k anyway. Our two goals in the subsequently described construction are 1) to reduce sufficiently the probability of this error occurring each time a decrement instruction is executed so that with probability 1 errors eventually stop occurring, and 2) to set up the simulation carefully so that it may recover from any finite number of errors.[10]

3.2 Simulating register machine

We will first describe how to modify M to obtain another register machine S that is easier for the CRD to simulate repeatedly to correct errors. There are two general modifications we make to M to generate S. The first consists of adding several instructions before M's first line and at the end (denoted line h below). The second consists of adding some instructions before every decrement of M in order to limit the number of decrements M is allowed to perform. We now describe these modifications in more detail.

Intuitively, S maintains a bound $b \in \mathbb{N}$ on the total number of decrements M is allowed to perform, and S halts if M exceeds this bound.[11] Although M halts if no errors occur, an erroneous jump may take M to a configuration unreachable from the initial configuration, so that M would not halt even if simulated correctly from that point on. To ensure that S always halts, it stores b in such a way that errors cannot corrupt its value. S similarly stores M's input n in an incorruptible way. Since we know in advance that M halts on input n but do not know how many steps it will take, the CRD will simulate S over and over again on the same input, each time incrementing the bound b, so that eventually b is large enough to allow a complete simulation of M on input n, assuming no errors.

If the original register machine M has m registers $r_1, \ldots, r_m$, then the simulating register machine S will have $m+4$ registers: $r_1, \ldots, r_m, r_{\text{in}}, r'_{\text{in}}, t, t'$. The first m registers behave exactly as in M, and the additional 4 registers will be used to help S maintain the input n and bound b.

The registers r_{in} and r'_{in} are used to store the input value n. The input value n is given to S, and initially stored in r_{in}, and passed to r'_{in} and r_1 before any other commands are executed. This allows the input n to be in register r_1 for use by M, while retaining the input value n in r'_{in} for the next execution of S. We want to enforce the invariant that, even if errors occur, $r_{\text{in}} + r'_{\text{in}} = n$, so that the value n can be restored to register r_{in} when S is restarted. To ensure that this invariant is maintained, the values of registers r_{in} and r'_{in} change only with one of the two following sequences (or equivalent CRN implementations of the sequence): $\texttt{dec}(r_{\text{in}},k)\texttt{;inc}(r'_{\text{in}})$ or $\texttt{dec}(r'_{\text{in}},k)\texttt{;inc}(r_{\text{in}})$.

S is simulated multiple times by the CRD. To make it easier for the CRD to "reset" S by simply setting the current instruction to be the first line, S assumes that the work registers $r_2, \ldots, r_m$ are

[10] If the error takes the register machine M to a configuration from which it halts but possibly produces the wrong answer, then, assuming (1) is accomplished by other means, it is easy to ensure (2): the CRD can simply simulate M over and over again in an infinite loop, always updating the CRD's output to be the most recent output of M. Since (1) ensures that errors eventually stop occurring, all but finitely many simulations of M give the correct answer, causing the CRD to stabilize on this answer. Most of the complexity of the construction described in the subsequent sections is to handle the case that the error takes M to a configuration from which it does not halt if simulated correctly from that point on.

[11] It is sufficient to bound the number of decrements, rather than total instructions, since we may assume without loss of generality that M contains no "all-increment" cycles. (If it does then either these lines are not reachable or M enters an infinite loop.) Thus any infinite computation of M must decrement infinitely often.

initially positive and must be set to 0. It sets r_1 to 0 before using r_{in} and r'_{in} to initialize r_1 to n. In both pairs of registers $r_{\text{in}}, r'_{\text{in}}$ and t, t', S sometimes needs to transfer the entire quantity stored in one register to the other. We describe below a "macro" for this operation, which we call `flush`. The following three commands will constitute the `flush`(r,r') command for any registers r and r'.

`flush`(r,r')	i:	`dec`$(r, i+3)$
	$i+1$:	`inc`(r')
	$i+2$:	`goto`(i)

While r has a non-zero value, it will be decremented and r' will be incremented. To flush from one register into more than one other register, we can increment multiple registers in the place of line $i+1$ above. We denote this macro as `flush`$(r,r'$ `and` $r'')$. Note, that if an error occurs in a `flush`, the invariant on the sum of the two registers will still be maintained.

To set register r to 0, we use the following macro, denoted `empty`(r).

`empty`(r)	i:	`dec`$(r, i+2)$
	$i+1$:	`goto`(i)

Finally, we introduce one additional macro, `move`(r,r',k), which decrements one register r and increments another r', unless the first register is 0, in which case neither of them changes and the machine jumps to line k. This will be useful when we want to argue that the sum $r + r'$ is constant despite error.

`move`(r,r',k)	i:	`dec`$(r, i+2)$
	$i+1$:	`inc`(r')

The macro `move`(t,t',h) is used by S to count down the "timer" register t (while storing its value in t'), in order to limit the number of decrements M is allowed, by inserting a `move`(t,t',h) instruction before every decrement of M, where line h will halt the computation.

Combining all techniques described above, the first instructions of S when simulating M will be as follows:

1:	`empty`(r_1)
2:	`empty`(r_2)
	$\vdots$
m:	`empty`(r_m)
$m+1$:	`flush`$(r'_{\text{in}}, r_{\text{in}})$
$m+2$:	`flush`$(r_{\text{in}}, r'_{\text{in}}$ `and` $r_1)$
$m+3$:	`flush`(t', t)
$m+4$:	first line of M

The first m lines ensure that registers $r_1, \ldots, r_m$ are initialized to zero. Lines $m+1$ and $m+2$ pass the input value n (stored as $r_{\text{in}} + r'_{\text{in}}$) to register r_1 (input register of M) while still saving the input value in r'_{in} to be reused the next time S is run. Line $m+3$ passes the full value of t' to t, so that the value of register t can be used to bound the number of decrements that M executes. After line $m+3$, S executes the commands of M, starting with the initial line of M (or, if the initial line of M is a decrement command, decrementing t before executing the first line of M).

Registers t and t' are used to record and bound the number of decrements performed by M, and their values are modified similarly to r_{in} and r'_{in} so that S maintains the invariant $t + t' = b$ throughout each execution. Before every decrement of M, we insert the instruction `move`(t,t',h). S uses lines h and $h+1$ added at the end to halt if M exceeds the decrement bound:

$$
\begin{aligned}
h\colon &\quad \texttt{inc}(t)\\
h+1\colon &\quad \texttt{reject}
\end{aligned}
$$

As S performs each decrement command in M, t is decremented (and t' is incremented). Thus, if the value of t is zero, then this means M has exceeded the allowable number of decrements, and computation halts (with a `reject`, chosen merely as a convention) immediately after incrementing t on line h. This final increment ensures that when the CRD simulates S again, the bound $b = t + t'$ will be 1 larger than it was during the previous execution of S.

Let d_n be the number of decrements M makes on input n without errors. If S is run without errors from an initial configuration (starting on line 1) with $r_{\text{in}} + r'_{\text{in}} = n$ and $t + t' \geq d_n$, then it will successfully simulate M. Since the invariants on $r_{\text{in}} + r'_{\text{in}} = n$ and the constant value of $t + t'$ are maintained throughout the computation (although $t + t'$ increases between each computation) — even in the face of errors — the only thing required to reset S after it halts is to set the current line to 1.

3.3 CRD simulation of the modified register machine

We now construct a CRD $\mathcal{D}$ to simulate S, while reducing the probability of an error each time an error could potentially occur. Besides the species described in Section 3.1, we introduce the following new species: voting species N and Y, an "accuracy" species A, and four "clock" species C_1, C_2, C_3, and C_4. The accuracy and clock species will be used to reduce the probability of error each time a decrement is simulated.

The initial state of $\mathcal{D}$ on input $n \in \mathbb{N}$ is $\{nR_{\text{in}}, 1L_1, 1Y, 1C_1\}$ — i.e., start with register $r_{\text{in}} = n$, initialize the register machine S at line 1, have initial vote "yes" (arbitrary choice), and start the "clock module" in the first of its four stages.

Recall that the only source of error in the CRD simulation is from the decrement instruction `dec(`r`,`k`)` when R is present, but the jump reaction $L_i \to L_k$ occurs instead of the decrement reaction $L_i + R \to L_{i+1}$. This would cause the CRD to erroneously perform a jump when it should instead decrement register r. To decrease the probability of this occurring, we can slow down the jump reaction, thus decreasing the probability of it occurring before the decrement reaction when R is present.

The following reactions, which we call the "clock module," implement a random walk that is biased in the reverse direction, so that C_4 is present sporadically, with the bias controlling the frequency of time C_4 is present. The count of "accuracy species" A controls the bias:

$$
\begin{aligned}
C_1 &\to C_2, &\quad C_2 + A &\to C_1 + A,\\
C_2 &\to C_3, &\quad C_3 + A &\to C_2 + A,\\
C_3 &\to C_4, &\quad C_4 + A &\to C_3 + A.
\end{aligned}
$$

We modify the conditional jump reaction to require a molecule of C_4 as a reactant:

`dec(`r`,`k`)`	$L_i + R \to L'_i$ $L'_i + C_4 \to L_{i+1} + C_1 + A$ $L_i + C_4 \to L_k + C_1 + A$

Increasing the count of species A decreases the expected time until the reaction $C_{i+1} + A \to C_i + A$ occurs, while leaving the expected time until reaction $C_i \to C_{i+1}$ constant. This has the effect that C_4 is present less frequently, delaying the conditional jump reaction.[12]

[12]Intuitively, with an ℓ-stage clock, if there are a molecules of A, the frequency of time that C_ℓ is present is less than $\frac{1}{a^{\ell-1}}$. A stage $\ell = 4$ clock is used to ensure that the error decreases quickly enough that with probability 1 a finite number of errors are ever made, and the last error occurs in finite expected time. A more complete analysis of the clock module is contained in Section 4.

We refer to the first reaction implementing `dec(`r`,`k`)` as the *decrement* reaction and the third as the *jump* reaction; note that if $\#R > 0$ then both reactions are possible so they probabilistically compete (the "correct" reaction in that case is the decrement, and the jump reaction executing instead would constitute a simulation error). An additional A molecule is produced to lower the probability of simulation error on the next decrement instruction. Each simulation of `dec(`r`,`k`)` converts a C_4 molecule to C_1, which restarts the clock. The reason we convert C_4 to C_1 is to ensure that the increase in $\#A$ has an immediate effect of lowering the probability of C_4 being present (this is explained in more detail in the proof of Theorem 4.1).

The `accept`, `reject`, `goto`, and `inc` commands cannot result in errors for the CRD simulation, so we keep their reactions unchanged from Section 3.1. Assuming the command occurs on line i, the reactions to simulate each command are given in the following table:

`accept`	$L_i \to H_Y$
`reject`	$L_i \to H_N$
`goto(`k`)`	$L_i \to L_k$
`inc(`r`)`	$L_i \to L_{i+1} + R$

After the CRD has completed the simulation and stabilizes, we would like the CRD to store the output of computation (either H_Y or H_N) and restart. At any time, there is precisely one molecule of either of the two voting species Y or N and none of the other, representing the CRD's "current vote." The vote is updated after each simulation of S, and S is reset to its initial configuration, via these reactions:

$$H_Y + Y \to L_1 + Y \tag{3.1}$$
$$H_Y + N \to L_1 + Y \tag{3.2}$$
$$H_N + Y \to L_1 + N \tag{3.3}$$
$$H_N + N \to L_1 + N \tag{3.4}$$

Finally, we give special CRN implementations of the "flush", "empty", and "move" macros explained in Section 3.2. Each of these involves simulating decrements (in addition to other instructions), so each decrement performed by each macro must follow the same rule explained above: increase the count of the species A, and "reset" the clock by consuming C_4 and producing C_1.

We use the following optimized implementation of the flush macro. Although we give register machine instructions that have an equivalent effect as the macro, think of the entire macro as a single line of register machine code. Hence if the macro appears on line i, assume the line after the macro is $i + 1$.[13]

`flush(`r`,`r'`)`	
i: `dec(`r`,`$i+1$`)`	$L_i + R \to L'_i + R'$
`inc(`r'`)`	$L'_i + C_4 \to L_i + C_1 + A$
`goto(`i`)`	$L_i + C_4 \to L_{i+1} + C_1 + A$

If we execute `flush(`r`,`r'` and `r''`)`, the extra increment is also done by the first reaction.

[13] Although there are three instructions and three reactions in this implementation of `flush(`r`,`r'`)`, there is not a 1-1 mapping between instructions and reactions; the three reactions collectively have the same effect as the three instructions, assuming the third reaction does not erroneously happen when the first reaction is possible.

flush(r,r' and r'')	
i: `dec`(r,$i+1$)	$L_i + R \to L'_i + R' + R''$
`inc`(r')	$L'_i + C_4 \to L_i + C_1 + A$
`inc`(r'')	$L_i + C_4 \to L_{i+1} + C_1 + A$
`goto`(i)	

The following is an optimized implementation of the empty macro.

empty(r)	
i: `dec`(r,$i+1$)	$L_i + R \to L'_i$
`goto`(i)	$L'_i + C_4 \to L_i + C_1 + A$
	$L_i + C_4 \to L_{i+1} + C_1 + A$

The following is an optimized implementation of the move macro.

move(r,r',k)	
i: `dec`(r,k)	$L_i + R \to L'_i + R'$
`inc`(r')	$L'_i + C_4 \to L_{i+1} + C_1 + A$
	$L_i + C_4 \to L_k + C_1 + A$

In Section 4, we prove that with probability 1, this CRD decides the predicate decided by M.

3.4 Example of construction of CRD from register machine

In this subsection we give an example of a register machine M that decides a predicate, a register machine S that simulates M as described in Section 3.2, and a CRD $\mathcal{D}$ that simulates S as described in Section 3.3. The register machine M is the same as in the example of Section 3.1, which decides the parity of input n.

The machine M requires only the input register r_1. Machine S has 5 registers: r_1, r, r', t, and t', and extra commands as described in Section 3.3. Finally, machine S can be simulated by a CRD $\mathcal{D}$ with reactions as described in Section 3.3. The initial state of $\mathcal{D}$ is $\{1L_1, 1Y, 1C_1, nR\}$. All three constructions for the example predicate are shown side-by-side in the following table. The commands in M are vertically aligned with their respective representations in S and the CRN. The line described as h in Section 3.2 is line 12 in the table.

In addition to reactions in the table, $\mathcal{D}$ requires the clock module reactions:

$$\begin{aligned} C_1 \to C_2, &\quad C_2 + A \to C_1 + A, \\ C_2 \to C_3, &\quad C_3 + A \to C_2 + A, \\ C_3 \to C_4, &\quad C_4 + A \to C_3 + A. \end{aligned}$$

M	S	$\mathcal{D}$
	1: $\texttt{empty}(r_1)$	$L_1 + R_1 \to L'_1$ $L'_1 + C_4 \to L_1 + C_1 + A$ $L_1 + C_4 \to L_2 + C_1 + A$
	2: $\texttt{flush}(r'_{\text{in}}\texttt{,}r_{\text{in}})$	$L_2 + R'_{\text{in}} \to L'_2 + R_{\text{in}}$ $L'_2 + C_4 \to L_2 + C_1 + A$ $L_2 + C_4 \to L_3 + C_1 + A$
	3: $\texttt{flush}(r_{\text{in}}\texttt{,}r'_{\text{in}}\ \texttt{and}\ r_1)$	$L_3 + R_{\text{in}} \to L'_3 + R'_{\text{in}} + R_1$ $L'_3 + C_4 \to L_3 + C_1 + A$ $L_3 + C_4 \to L_4 + C_1 + A$
	4: $\texttt{flush}(t'\texttt{,}t)$	$L_4 + T' \to L'_4 + T$ $L'_4 + C_4 \to L_4 + C_1 + A$ $L_4 + C_4 \to L_5 + C_1 + A$
1: $\texttt{dec}(r_1\texttt{,5})$	5: $\texttt{move}(t\texttt{,}t'\texttt{,12})$	$L_5 + T \to L'_5 + T'$ $L'_5 + C_4 \to L_6 + C_1 + A$ $L_5 + C_4 \to L_{12} + C_1 + A$
	6: $\texttt{dec}(r_1\texttt{,11})$	$L_6 + R_1 \to L'_6$ $L'_6 + C_4 \to L_7 + C_1 + A$ $L_6 + C_4 \to L_{11} + C_1 + A$
2: $\texttt{dec}(r_1\texttt{,4})$	7: $\texttt{move}(t\texttt{,}t'\texttt{,12})$	$L_7 + T \to L'_7 + T'$ $L'_7 + C_4 \to L_8 + C_1 + A$ $L_7 + C_4 \to L_{12} + C_1 + A$
	8: $\texttt{dec}(r_1\texttt{,10})$	$L_8 + R_1 \to L'_8$ $L'_8 + C_4 \to L_9 + C_1 + A$ $L_8 + C_4 \to L_{10} + C_1 + A$
3: `goto(1)`	9: `goto(5)`	$L_9 \to L_5$
4: `accept`	10: `accept`	$L_{10} \to H_Y$
5: `reject`	11: `reject`	$L_{11} \to H_N$
	12: $\texttt{inc}(t)$	$L_{12} \to L_{13} + T$
	13: `reject`	$L_{13} \to H_N$
	update vote and reset S	$H_Y + Y \to L_1 + Y$ $H_Y + N \to L_1 + Y$ $H_N + Y \to L_1 + N$ $H_N + N \to L_1 + N$

4 Correctness of simulation

In this section we show that the construction of Section 3 is correct. We achieve probability 1 computation in finite expected time, and indeed polynomially related to the number of steps of the register machine computation.

Theorem 4.1. *Let $\mathcal{D}$ be the CRD described in Section 3, simulating an arbitrary register machine M deciding predicate $\phi : \mathbb{N} \to \{0,1\}$. Then $\mathcal{D}$ decides ϕ with probability 1. Furthermore, the expected time until $\mathcal{D}$ stabilizes to the correct answer is $O(t_n^8)$, where t_n is the number of steps M takes to halt on input n.*

Proof. We will show three things. First, so long as S is repeatedly simulated by $\mathcal{D}$, if a finite number of errors occur, then eventually it correctly simulates M (and therefore $\mathcal{D}$'s voters eventually are updated to the correct value and stay there). Second, with probability 1, $\mathcal{D}$ makes only a finite number of errors in simulating S. Finally, we bound the expected time to stabilize to the correct answer.

Let $n \in \mathbb{N}$ be the input to M. We note that the CRD simulation of S may experience errors, but the errors take a very particular form. If $\mathcal{D}$ makes an error, it is because the first and third reactions below (corresponding to the two branches a decrement instruction might follow) are possible, but executing the third is an error:

$$\begin{aligned} L_i + R &\to L_i' \\ L_i' + C_4 &\to L_{i+1} + C_1 + A \\ L_i + C_4 &\to L_k + C_1 + A \end{aligned}$$

(This applies similarly if the decrement is on registers r_{in}, r'_{in}, t, or t', or to similar reactions that "test species for 0" such as those in $\texttt{empty}(r_{\text{in}})$, $\texttt{flush}(r_{\text{in}},r'_{\text{in}})$ $\texttt{move}(t,t',h)$.) This corresponds to the command "in line i, decrement register j and go to line $i+1$, unless the register is 0, in which case go to line k instead of $i+1$." The error in simulation can be captured by a modified model of a register machine in which the following error may occur: when doing this decrement instruction, even if register j is positive, it is possible for the machine to nondeterministically choose to jump to line k without decrementing register j.

Even in the face of such errors, S maintains the following useful invariants, which follow by inspection of the reactions implementing S.

1. $r_{\text{in}} + r'_{\text{in}} = n$

2. If S has been simulated by $\mathcal{D}$ several times, and in N of these simulations, S's simulation of M has "timed out" (the register t decremented to 0 before the simulation of M was complete), then on the next simulation of S, we will have $t + t' = N + 1$. (In the example in Section 3.4, this is due to line 12 that increments t, without changing t', if an attempted $\texttt{move}(t,t',12)$ ever failed.)

The above are easy to check by inspection of the machine S. (The example in Section 3.4 is helpful).

Since the initial instructions of S (assuming they are executed without error) reset registers r'_{in}, t', and $r_2, \ldots, r_m$ to 0, and they reset r_1 to be its correct initial value n, *any* configuration of S such that

1. the current line is the first,

2. $r_{\text{in}} + r'_{\text{in}} = n$, and

3. $t + t'$ is at least the total number of decrements M will do on input n

is a "correct" initial configuration, in the sense that an error-free execution of S from that configuration will result in the same answer as an error-free execution from the original initial configuration (in which r'_{in}, t', and $r_2, \ldots, r_m$ start out equal to 0). Since S maintains (2) and (3) even in the face of errors, this means that to reset S, $\mathcal{D}$ needs only to consume the "halt molecule" H_Y or H_N and produce L_1, which is precisely what is done by reactions (3.1), (3.2), (3.3), and (3.4).

Therefore, once $\mathcal{D}$ eventually stops making errors in simulating S, then S faithfully simulates M (from some configuration possibly unreachable from the input), for at most $t + t'$ decrements in M, at which point $\mathcal{D}$ resets S. At that point, $\mathcal{D}$ simulates S repeatedly, and after each time that S stops its simulation of M because the t register is depleted, the sum $t + t'$ is incremented.

Therefore the sum $t + t'$ eventually becomes large enough for S to simulate M completely. At that point the voters are updated to the correct value and stay there forever.

It remains to show that $\mathcal{D}$ is guaranteed to make a finite number of errors in simulating S. Recall that the only error possible is, on a decrement instruction, to execute a "jump" even when a register is positive. Each time that a decrement instruction is reached, the count of molecule A is incremented (whether the decrement or the jump occurs). The "clock" module implements a biased random walk on the state space $\{C_1, C_2, C_3, C_4\}$ with rate 1 to increment the subscript and rate equal to the count of A — denoted by ℓ — to decrement the index. If this Markov process is in equilibrium, the probability of the process being in state C_4 is $\frac{1}{\ell^3}$ (see [16] for example). Since the process starts in state C_1, every nonequilibrium distribution reached at any time after starting assigns *lesser* probability to state C_4, so the probability of the process being in state C_4 is *at most* $\frac{1}{\ell^3}$.

Thus, when simulating a decrement $\texttt{dec}(r,k)$ (on line i), if there is at least one R molecule, the probability that L_i encounters C_4 prior to encountering R is at most $\frac{1}{\ell^3}$. At each decrement instruction (whether the decrement or the jump happens), A's count is incremented, so for all $\ell \in \mathbb{Z}^+$, the probability of an error on the ℓ'th decrement instruction (counting from the beginning of the entire execution of $\mathcal{D}$; not just from the beginning of one simulation of S) is at most $\frac{1}{\ell^3}$. Since $\sum_{\ell=1}^{\infty} \frac{1}{\ell^3} < \infty$, by the Borel-Cantelli Lemma, the probability that only finitely many errors occur is 1.[14] (In fact, error probability $\frac{1}{\ell^2}$ would have sufficed here, but the lower error probability of $\frac{1}{\ell^3}$ will help us derive a finite expected time until errors cease.)

We now derive a bound on the expected time for the register machine simulation to converge to the correct output. We first establish a bound on the expected total number of *steps* (of the simulated register machine), and then use this to bound the expected *time*. For each ℓ, let P_ℓ be the probability that the *last* error occurs on the ℓ'th decrement instruction. The probability that an error occurs on the ℓ'th decrement instruction is at most $1/\ell^3$, so $P_\ell \leq \frac{1}{\ell^3}$. Then, the expected step (measured over all decrement instructions, not over all steps of the register machine) of the last error is at most $\sum_{\ell=1}^{\infty} \ell \cdot \frac{1}{\ell^3} = \frac{\pi^2}{6}$. Hence we have only a constant expected number of steps until the last error.

Because of this, the expected number of steps to stabilization is at most $t_e + t_t + t_n$, where $t_e = O(1)$ is the expected number of simulated steps until the last error happens, t_n is the expected number of steps for one simulation of the register machine M to complete, given input n and assuming no errors happen, and t_t is the expected number of steps until the timer T reaches to at least t_n . Since the timer increases by 1 each simulation, $t_t = O(t_n^2)$, so the expected number of steps until stabilization is at most $t_e + t_t + t_n = O(1) + O(t_n^2) + t_n = O(t_n^2)$.

Now we use this bound on the expected number of steps to bound the expected *real time* until stabilization. Each decrement increases the "accuracy" species A. The slowest reaction in the system is a "jump" of the form $L_i + C_4 \to L_k + C_1 + A$. After ℓ instructions, the count of A is at most ℓ (assuming in the worst case that every executed instruction is a decrement). Lemma A.4 (or Lemma 8 in the journal version) of [16] shows that with a clock module having s stages, the expected time for the ℓ'th decrement to occur is at most $O(\ell^{s-1})$, which is $O(\ell^3)$ since $s = 4$ in our case.[15] Therefore, all t_n^2 steps take total expected time $O(\sum_{\ell=1}^{O(t_n^2)} \ell^3) = O(t_n^8)$. □

The proof of Theorem 4.1 assumes a constant volume. Suppose instead we obey the finite density constraint — discussed in Section 2 — by assuming the volume grows to remain proportional to the

[14] Recall that the Borel-Cantelli Lemma does not require the events to be independent, and indeed they are not in our case (e.g., failing to decrement may increase or decrease the chance of error on subsequent decrement instructions.

[15] They actually show it is $O(\ell^{s-1}v/k)$, where v is the volume (1 in our case), k is the rate constant on all reactions (also 1 in our case).

current molecular count. Note that if the volume is increased, the rates of bimolecular reactions decrease while the rates of unimolecular reactions remain unchanged. To avoid possibly increasing the error probability, we can simply change all unimolecular reactions to be bimolecular with a catalyst F that has constant count 1. For example, $C_i \to C_{i+1}$ becomes $C_i + F \to C_{i+1} + F$. In volume 1, this new system behaves identically to the original, since the propensities have not changed at all. However, since every reaction is now bimolecular, although the *absolute* rate of each reaction decreases as the volume increases, the *relative* rate between any bimolecular reactions (i.e., each reaction's probability of being the next reaction to occur from any given configuration) is identical to the volume 1 case. Thus, the new system with a changing volume behaves identically to the original system with a fixed volume, except that the expected time per computation step now increases with the volume.

How does the time analysis change if the finite density constraint is imposed? The expected time for a step of the register machine simulation is $O(\#A^3 \cdot v)$ (by Lemma A.4 of [16]). If l steps have previously occurred, $\#A = O(\ell)$ and $v = O(\ell)$. (Note that the volume is either dominated by A or by some register species, but at most one A or one register molecule is created per step.) Thus the expected time for the ℓth step increases from ℓ^3 to ℓ^4 with the finite density constraint. The expected time until stabilization increases from $O(\sum_{\ell=1}^{O(t_n^2)} \ell^3) = O(t_n^8)$ to $O(\sum_{\ell=1}^{O(t_n^2)} \ell^4) = O(t_n^{10})$.

5 Limit computable predicates

In this section we extend the technique of Section 3 to show that every Δ_2^0 (limit-computable) predicate is decidable with probability 1 by a CRD (Theorem 5.1). We also show the converse result (Theorem 5.2) that every predicate decidable with probability 1 by a CRD is in Δ_2^0. Theorems 5.1 and 5.2 give our main result, that probability 1 decidability by CRDs is exactly characterized by the class Δ_2^0.

Theorem 5.1. *Every Δ_2^0 predicate is decidable with probability 1 by a CRD.*

Proof. For every Δ_2^0 predicate $\phi : \mathbb{N} \to \{0, 1\}$, there is a computable function $r : \mathbb{N}\times\mathbb{N} \to \{0, 1\}$ such that, for all $n \in \mathbb{N}$, $\lim_{t\to\infty} r(n, t) = \phi(n)$. Therefore there is a register machine M that computes r. As in Section 3, we first construct a register machine S that simulates M in a controlled fashion that ensures errors in simulation by a CRD will be handled gracefully.

We now argue that the definition of Δ_2^0 does not change if we require the register machine M computing $r(n, t)$ to halt in exactly $2t$ steps. To see this, let M'' be a register machine that computes r. Define the non-halting register machine M' that, on input n, does the following. In an infinite loop, for $t' = 1, 2, 3, \ldots$, $M'(n)$ runs $M''(n, t')$, updating its own output to be the output of $M''(n, t')$ for the most recent value of t' for which $M''(n, t')$ produced an output. Define the halting register machine M that, on input n, t (each input stored in its own register r_1 and r_2, respectively), does the following. $M(n, t)$ alternately decrements r_2 and then executes the next instruction of $M'(n)$, updating its output to match the current output of M', and halting when r_2 reaches 0. Clearly, $M(n, t)$ halts in exactly $2t$ steps (or $2t + 1$ if one counts the final halt as its own step). Also, as $t \to \infty$, the largest value of t' for which $M'(n)$ simulates $M''(n, t')$ also approaches ∞, which by the definition of r implies that for sufficiently large t, $M(n, t)$ is the correct output.

Similar to Section 3, S uses a "timer" to simulate $M(n, t)$ for at most $2t$ steps. Unlike in Section 3, S decrements the timer after every step (not just the decrement steps) and the timer is incremented by 2 after each execution (in the previous construction, the timer is incremented by 1 and only if M exceeds the time bound). Note that no matter what errors occur, no execution can go for longer than $2t$ steps by the timer construction in Section 3. So S will dovetail the computation as before, running $M(n,1)$ for 2 steps, $M(n,2)$ for 4 steps, $M(n,3)$ for 6 steps, etc., and in between each execution of $M(n,t)$, update its voting species with the most recent answer.

As in the construction of Section 3, so long as the ℓ'th decrement has error probability at most $\frac{1}{\ell^3}$, then by the Borel-Cantelli lemma, with probability 1 a finite number of errors occur. Errors in the CRD simulating S maintain the input without error and increment the timer value without error. Thus after the last error occurs, and after t is sufficiently large that $r(n,t) = \phi(n)$, the CRD will stop updating its voter, and the CRD's output will be correct. □

The next theorem shows that *only* Δ_2^0 predicates are decidable with probability 1 by a CRD.

Theorem 5.2. *Let the CRD $\mathcal{D}$ decide predicate $\phi : \mathbb{N} \to \{0,1\}$ with probability 1. Then $\phi \in \Delta_2^0$.*

Proof. It suffices to show that there is a computable function $r : \mathbb{N} \times \mathbb{N} \to \{0,1\}$ such that, for all $n \in \mathbb{N}$, $\lim_{t\to\infty} r(n,t) = \phi(n)$. The function r is defined for $n, t \in \mathbb{N}$ as $r(n,t) = 1$ if the probability that $\mathcal{D}$ is outputting "yes" after exactly t reactions have occurred is at least the probability that $\mathcal{D}$ is outputting "no," when started in an initial state encoding n, and $r(n,t) = 0$ otherwise.

To see that r is computable, consider the Turing machine M that, on input (n,t), searches the set of states reachable from the initial state $\mathbf{i}$ of $\mathcal{D}$ (with n copies of its input species) by executing exactly t reactions. Call t the *depth* of the state; we consider identical states that are encountered at different depths of the search differently. If $\mathbf{c}$ is reachable from $\mathbf{i}$ in exactly t reactions, then write $\mathbf{c} \in \mathrm{REACH}^t(\mathbf{i})$.

Each (state,depth) pair $(\mathbf{c}, t)$ is assigned a measure $\mu : \mathbb{N}^\Lambda \times \mathbb{N} \to [0,1]$ recursively as follows.[16] The initial state $(\mathbf{i}, 0)$ has $\mu(\mathbf{i}, 0) = 1$, and all other states $\mathbf{c}$ have $\mu(\mathbf{c}, 0) = 0$. If state $\mathbf{c}$ is reachable by exactly 1 reaction from exactly ℓ states $\mathbf{c}_1, \ldots, \mathbf{c}_\ell$, each of which are reachable from $\mathbf{i}$ by exactly $t-1$ reactions, with $\mathbf{c}_j$ followed by a single reaction that occurs as the next reaction in state $\mathbf{c}_j$ with probability p_j to reach state $\mathbf{c}$, then

$$\mu(\mathbf{c}, t) = \sum_{j=1}^{\ell} p_j \cdot \mu(\mathbf{c}_j, t-1).$$

That is, $\mu(\mathbf{c}, t)$ is the probability that after exactly t reactions have occurred from the initial state $\mathbf{i}$, $\mathcal{D}$ is in state $\mathbf{c}$. Each value p_j is a rational number computable from the counts of species in state $\mathbf{c}_j$, so $\mu(\mathbf{c}, t)$ is computable.

For $b \in \{0,1\}$, let

$$\mu_b(t) = \sum_{\substack{\mathbf{c} \in \mathrm{REACH}^t(\mathbf{i}) \\ \Phi(\mathbf{c}) = b}} \mu(\mathbf{c}, t).$$

That is, $\mu_b(t)$ is the probability that, after exactly t reactions, the output of $\mathcal{D}$ is b. If $\mathcal{D}$ decides ϕ with probability 1, then on input n with $\phi(n) = b$, we have $\lim\limits_{t\to\infty} \mu_b(t) = 1$ and $\lim\limits_{t\to\infty} \mu_{1-b}(t) = 0$.

M computes $\mu_0(t)$ and $\mu_1(t)$ and outputs 1 if $\mu_1(t) \geq \mu_0(t)$ and outputs 0 otherwise. Due to the limit stated above, for all sufficiently large t, $M(n,t)$ outputs $\phi(n)$, whence $\phi \in \Delta_2^0$. □

[16] Technically, we are defining, for each $t \in \mathbb{N}$, a measure on the set of all states, giving the state's probability of being reached in exactly t steps, so for each $t \in \mathbb{N}$, $\mu(\cdot, t) : \mathbb{N}^\Lambda \to [0,1]$ is a probability measure on $\mathbb{N}^\Lambda$. Since the evolution of the system is Markovian, once we know the probability $\mu(\mathbf{c}, t)$ of ending up in state $\mathbf{c}$ after precisely t steps, it does not matter the particular sequence of $t-1$ states preceding $\mathbf{c}$ that got the system there, in order to determine probability of the various states that could follow $\mathbf{c}$.

6 Computing functions

There is a natural notion of the computation of a *function* $f : \mathbb{N} \to \mathbb{N}$ with CRNs [5], in which the *count* of some "output species" encodes the output $f(n)$. Since a predicate $\phi : \mathbb{N} \to \{0, 1\}$ has a finite output range, it is possible for the identity of a single molecule (Y or N) to encode the output. In the construction of Section 3, this makes it possible for a single reaction (one of reactions (3.1)-(3.4)) to compare the CRD's output to the most recent register machine output and change the CRD's output only if they disagree. After errors stop, the only "output-checking" reactions that occur are (3.1) and (3.4), which use the CRD output molecule only as a catalyst, so the output remains stable. If the output is instead represented by a count of some species, comparison of the counts of two species necessarily involves *consuming* them, so it is not obvious how to do this comparison without altering the count of the output species.

This section describes how to construct a chemical reaction network that can compute an arbitrary Turing-computable function $f : \mathbb{N} \to \mathbb{N}$ with probability 1.[17] We first formally define the semantics of such CRNs. A *chemical reaction computer (CRC)* is a tuple $\mathcal{C} = (\Lambda, R, X, Y, \sigma)$, where (Λ, R) is a CRN, $X \in \Lambda$ is the *input species*, $Y \in \Lambda$ is the *output species*, and $\sigma \in \mathbb{N}^{\Lambda \setminus \{X\}}$ is the *initial context*.[18] As with CRD's, we require initial configurations $\mathbf{x}$ of $\mathcal{C}$ with input $n \in \mathbb{N}$ to obey $\mathbf{x}(X) = n$ and $\mathbf{x}(S) = \sigma(S)$ for all $S \in \Lambda \setminus \{X\}$, calling them *valid initial configurations*. Given a configuration $\mathbf{c}$, the *output* $\Phi(\mathbf{c})$ of $\mathbf{c}$ is $\mathbf{c}(Y)$. Let $\mathcal{E} = (\mathbf{c}_0, \mathbf{c}_1, \ldots)$ be an infinite execution sequence of $\mathcal{C}$. We say that $\mathcal{E}$ has a defined output if there exists $m \in \mathbb{N}$ such that, for all but finitely many $i \in \mathbb{N}$, $\Phi(\mathbf{c}_i) = m$.[19] In this case, write $\Phi(\mathcal{E}) = m$; otherwise, let $\Phi(\mathcal{E})$ be undefined. If $n \in \mathbb{N}$, write $\mathcal{E}(\mathcal{C}, n)$ to denote the random variable representing an execution sequence of $\mathcal{C}$ on input n under the stochastic model defined in Section 2.4. We say that $\mathcal{C}$ *computes* $f : \mathbb{N} \to \mathbb{N}$ *with probability 1* if, for all $n \in \mathbb{N}$, $\Pr[\Phi(\mathcal{E}(\mathcal{C}, n)) = f(n)] = 1$.

The intuition behind our construction is as follows. As before, we run register machine simulations repeatedly, decreasing the probability of error after each simulated decrement. After each run, we compare the register machine output of the current run with the global output held in the amount of Y. If the amounts do not match, we then use the current machine output to replace the global output. By the Borel-Cantelli Lemma, at some point simulation errors stop, and the CRC then stabilizes on the correct output. Importantly, we must be careful to do the comparison in such a way that we do not affect the value of Y if it is already correct. The only way to do this is to compare an additional backup copy of Y.

We must be careful to manipulate the current run output and the global output Y, and their backup copies used in comparison, in such a way that errors cannot confound convergence.[20] Besides the macros defined previously, we will need a way to empty more than one register at a time such that a particular invariant is maintained. Suppose we have three registers y, y_{copy}, and y'_{copy}, and we want to maintain that invariant that $y = y_{\text{copy}} + y'_{\text{copy}}$. We occasionally want to empty these registers, but maintain the invariant even if we make a simulation error in decrementing them to 0. The following reactions implement this macro on line i:

[17] As in Section 3, we focus on functions taking single integers as input for the sake of simplifying the discussion. The ideas carry through just as easily to functions $f : \mathbb{N}^k \to \mathbb{N}$. One could modify the construction to simply have extra input registers for a direct encoding, or one could encode several inputs $n_1, \ldots, n_k \in \mathbb{N}$ into a single integer $n \in \mathbb{N}$ using some injective encoding function $e : \mathbb{N}^k \to \mathbb{N}$, which could be decoded as the first step of the register machine computation.

[18] For convenience we are assuming a single input species, so that once we know the initial context of $\mathcal{C}$, we can equivalently fully describe the input to $\mathcal{C}$ as a single natural number n describing the initial count of X.

[19] Equivalently, $\lim_{i \to \infty} \Phi(\mathbf{c}_i) = m$.

[20] An erroneous value of Y can be made permanent if its backup is actually correct — since we only compare the backup. Thus we must ensure that even in the presence of errors, certain invariants are maintained.

`empty(`y `and (`y_{copy} `or` y'_{copy}`))`
$L_i + Y \to L'_i$
$L'_i + Y_{\text{copy}} \to L''_i$
$L'_i + Y'_{\text{copy}} \to L''_i$
$L''_i + C_4 \to L_i + C_1 + A$
$L_i + C_4 \to L_{i+1} + C_1 + A$

Note that the correctness of the reactions require that $y = y_{\text{copy}} + y'_{\text{copy}}$ initially; in this case, if the first reaction happens, then $\#Y_{\text{copy}} + \#Y'_{\text{copy}} > 0$, so either the second or the third reaction is guaranteed to be possible.

We now describe how a CRC can compute an arbitrary Turing computable function (any function computed by some register machine). We will use a similar construction to that described in Section 3. M is the name of the register machine that computes f, with S simulating M as in Section 3 (except for the modifications described in this section). Assume the register machine M does not have accept/reject semantics, and that it has an additional register r_{out} to store its output, represented by species R_{out}. Since neither S nor the CRC has accept/reject semantics, we do not require the special reactions (3.1), (3.2), (3.3), and (3.4) to transfer S's output to the CRC. S therefore is designed to run forever, and the CRC simply simply simulates it once.

The first instructions of S before simulating M are similar to those in Section 3. Assuming r_1 is the input register of M, $r_2, \ldots, r_m$ are the work registers of M, r_{out} is the output register of M, and the value $r_{\text{in}} + r'_{\text{in}}$ is always equal to the input value n:

1:	`empty(`r_1`)`
2:	`empty(`r_2`)`
	$\vdots$
m:	`empty(`r_m`)`
$m+1$:	`empty(`r_{out}`)`
$m+2$:	`flush(`r'_{in}`,`r_{in}`)`
$m+3$:	`flush(`r_{in}`,`r'_{in} `and` r_1`)`
$m+4$:	`flush(`t'`,`t`)`
$m+5$:	first line of M

S will execute `move(`t`,`t'`,`h`)` before every decrement of M, and line h is `inc(`t`)` followed by `goto(1)` (i.e., restart the whole simulation, after increasing the bound on the number of decrements M is allowed).

In addition to the registers described in Section 3, S will also have registers r^1_{out}, r^2_{out}, y_{copy}, y'_{copy}, and y, represented by species $R^1_{\text{out}}, R^2_{\text{out}}, Y_{\text{copy}}, Y'_{\text{copy}}, Y$. Y is the output species of the CRC; therefore, we must take care that its value stabilizes.

The species Y_{copy} and Y'_{copy} together (through the sum of their counts) will represent a copy of the output of the CRC; we will maintain the invariant that at all times, $\#Y = \#Y_{\text{copy}} + \#Y'_{\text{copy}}$. This will be used to compare the most recently computed output of S to the current output $\#Y$ of the CRC (indirectly, by comparing to $\#Y_{\text{copy}}$), changing the count of Y only if they disagree. The comparison empties r_{out}; however, after this its value will be needed if the comparison reports "false". So prior to the comparison, S copies the value of r_{out} into two registers r^1_{out} and r^2_{out}. S then compares r^1_{out} to y_{copy} and, if they are unequal, empties y, y_{copy}, and y'_{copy} and copies the value of r^2_{out} into y and y_{copy}.

To implement the above ideas, the following instructions are added after S is done with the simulation of M. Assume z is the last line of M.

prepare registers for comparison

$z+1$: `empty(`r^1_{out}`)`
$z+2$: `empty(`r^2_{out}`)`
$z+3$: `flush(`r_{out}`,`r^1_{out}` and `r^2_{out}`)`
$z+4$: `flush(`y'_{copy}`,`y_{copy}`)`
if $r^1_{\text{out}} = y_{\text{copy}}$*, restart at line 1, else go to line* $z+9$
$z+5$: `dec(`r^1_{out}`,`$z+8$`)`
$z+6$: `move(`y_{copy}`,`y'_{copy}`,`$z+9$`)`
$z+7$: `goto(`$z+5$`)`
$z+8$: `move(`y_{copy}`,`y'_{copy}`,1)`

If lines $z+5$ through $z+8$ detect that $r^1_{\text{out}} = y_{\text{copy}}$, then S is restarted at line 1. Otherwise, $r^1_{\text{out}} \neq y_{\text{copy}}$, and S goes to line $z+9$ below, which sets the counts of Y_{copy}, Y'_{copy}, and Y to 0, while maintaining the invariant that $\#Y = \#Y_{\text{copy}} + \#Y'_{\text{copy}}$ even if a simulation error occurs. Then, S flushes the value stored in the output register r^2_{out} into both Y_{copy} and Y. This maintains the invariant $\#Y = \#Y_{\text{copy}} + \#Y'_{\text{copy}}$ since a single reaction in the `flush(`r^2_{out}`,`y_{copy}` and `y`)` macro produces copies of Y_{copy} and Y simultaneously. The computation then restarts at line 1.

$z+9$: `empty(`y` and (`y_{copy}` or `y'_{copy}`))`
$z+10$: `flush(`r^2_{out}`,`y_{copy}` and `y`)`
$z+11$: `goto(1)`

There is a natural extension of the definition of "limit computable" from predicates to functions. We say that a function $f : \mathbb{N} \to \mathbb{N}$ is *limit computable* if there is a computable function $r : \mathbb{N}\times\mathbb{N} \to \mathbb{N}$ such that, for all $n \in \mathbb{N}$, $\lim\limits_{t\to\infty} r(n,t) = f(n)$.

Theorem 6.1. *A function $f : \mathbb{N} \to \mathbb{N}$ is computable with probability 1 by a CRC if and only if it is limit computable.*

Proof. The implication "limit computable" $\implies$ "computable with probability 1 by a CRC" follows by a straightforward modification of the proof of Theorem 5.1, using the simulation described in this section to simulate a register machine computing $r(n,t)$ on larger and larger values of t, where $\lim\limits_{t\to\infty} r(n,t) = f(n)$.

The reverse direction is similar to the proof of Theorem 5.2. The main difference is that there are more than two possible outputs. Let f be computed with probability 1 by the CRC $\mathcal{C}$, with output species Y. Let $\#_{n,t}Y$ denote the random variable representing the count of Y after exactly t reactions have occurred from the initial configuration representing input n.

We define $r : \mathbb{N}\times\mathbb{N} \to \mathbb{N}$ for all $n,t \in \mathbb{N}$ by

$$r(n,t) = \min\left\{\ k \in \mathbb{N} \ \middle|\ (\forall m \in \mathbb{N})\ \Pr[\#_{n,t}Y = k] \geq \Pr[\#_{n,t}Y = m]\ \right\},$$

i.e., $r(n,t)$ is the integer most likely to be the count of Y after exactly t reactions have occurred, breaking ties by choosing the smallest such integer. r is computable using the same bread-first search of the reachability graph of $\mathcal{C}$ as in the proof of Theorem 5.2.[21] Since $\mathcal{C}$ computes f with probability 1, $\lim\limits_{t\to\infty} \Pr[\#_{n,t}Y = f(n)] = 1$, which implies that $\lim\limits_{t\to\infty} r(n,t) = f(n)$, showing that f is limit computable. □

[21] The definition of $r(n,t)$ appears to require checking an infinite number of possible natural numbers m. However, only finitely many such natural numbers can have nonzero probability of being the count of Y after exactly t reactions since there are only a finite number of execution sequences of length t, each terminating in a configuration accounting for one possible value of $\#_{n,t}Y$. Therefore calculating $r(n,t)$ requires only searching through this finite list and picking the smallest natural number that has maximum probability.

7 Conclusion

Some open questions stand out.

Soloveichik, Cook, Winfree, and Bruck [16] showed how a CRN can simulate a Turing machine with only a polynomial slowdown, although the simulation is correct with probability strictly less than 1. Our probability 1 simulation of a register machine has a polynomial slowdown, but for some computations a register machine is necessarily exponentially slower than a Turing machine.[22]

Question 7.1. *Is it possible to compute any Turing computable predicate/function by a CRN with probability 1, which stabilizes in expected time polynomial in the speed of a Turing machine computing the same predicate/function?*

The next question was mentioned in Section 1.

Question 7.2. *Is there a "natural" characterization of probability 1 computation by CRNs in which the predicates computable by such CRNs are exactly the Turing computable predicates?*

Acknowledgements. We thank Shinnosuke Seki, Chris Thachuk, and Luca Cardelli for many useful and insightful discussions.

References

[1] Dana Angluin, James Aspnes, and David Eisenstat. Stably computable predicates are semilinear. In *PODC 2006: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 292–299, New York, NY, USA, 2006. ACM Press.

[2] James Aspnes and Eric Ruppert. An introduction to population protocols. *Bulletin of the European Association for Theoretical Computer Science*, 93:98–117, 2007.

[3] Robert Brijder. Output stability and semilinear sets in chemical reaction networks and deciders. In *DNA 2014: Proceedings of The 20th International Meeting on DNA Computing and Molecular Programming*, Lecture Notes in Computer Science. Springer, 2014.

[4] Luca Cardelli. Strand algebras for DNA computing. *Natural Computing*, 10(1):407–428, 2011.

[5] Ho-Lin Chen, David Doty, and David Soloveichik. Deterministic function computation with chemical reaction networks. *Natural Computing*, 13(4):517–534, 2014. Preliminary version appeared in DNA 2012.

[6] Yuan-Jyue Chen, Neil Dalchau, Niranjan Srinivas, Andrew Phillips, Luca Cardelli, David Soloveichik, and Georg Seelig. Programmable chemical controllers made from DNA. *Nature Nanotechnology*, 8(10):755–762, 2013.

[7] Matthew Cook, David Soloveichik, Erik Winfree, and Jehoshua Bruck. Programmability of chemical reaction networks. In Anne Condon, David Harel, Joost N. Kok, Arto Salomaa, and Erik Winfree, editors, *Algorithmic Bioprocesses*, pages 543–584. Springer Berlin Heidelberg, 2009.

[22] For a register machine even to meaningfully "read" an input n requires decrementing the input register at least n times to get it to 0, whereas a Turing machine can process n's binary representation in only $\approx \log n$ steps.

[8] William Feller. *An Introduction to Probability Theory and Its Applications*, volume 1. Wiley, January 1968.

[9] Daniel T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *Journal of Physical Chemistry*, 81(25):2340–2361, 1977.

[10] Richard M Karp and Raymond E Miller. Parallel program schemata. *Journal of Computer and system Sciences*, 3(2):147–195, 1969.

[11] Ernst W. Mayr. An algorithm for the general Petri net reachability problem. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, STOC '81, pages 238–246, New York, NY, USA, 1981. ACM.

[12] Carl A Petri. Communication with automata. Technical report, DTIC Document, 1966.

[13] Hartley Rogers, Jr. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill Series in Higher Mathematics. McGraw-Hill Book Company, New York-St. Louis-San Francisco-Toronto-London-Sydney-Hamburg, 1967.

[14] Joseph R. Shoenfield. On degrees of unsolvability. *Annals of Mathematics*, 69(3):644–653, 1959.

[15] Robert Irving Soare. Interactive computing and relativized computability. *Computability: Turing, Gödel, Church, and beyond (eds. BJ Copeland, CJ Posy, and O. Shagrir), MIT Press, Cambridge, MA*, pages 203–260, 2013.

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

[17] 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, 2010. Preliminary version appeared in DNA 2008.

[18] Vito Volterra. Variazioni e fluttuazioni del numero dindividui in specie animali conviventi. *Mem. Acad. Lincei Roma*, 2:31–113, 1926.

[19] Gianluigi Zavattaro and Luca Cardelli. Termination problems in chemical kinetics. *CONCUR 2008-Concurrency Theory*, pages 477–491, 2008.