CRN++: Molecular Programming Language

Marko Vasic, David Soloveichik, and Sarfraz Khurshid

The University of Texas at Austin, USA
{vasic,david.soloveichik,khurshid}@utexas.edu

Abstract. Synthetic biology is a rapidly emerging research area, with expected wide-ranging impact in biology, nanofabrication, and medicine. A key technical challenge lies in embedding computation in molecular contexts where electronic micro-controllers cannot be inserted. This necessitates effective representation of computation using molecular components. While previous work established the Turing-completeness of chemical reactions, defining representations that are faithful, efficient, and practical remains challenging. This paper introduces *CRN*++, a new language for programming deterministic (mass-action) chemical kinetics to perform computation. We present its syntax and semantics, and build a compiler translating *CRN*++ programs into chemical reactions, thereby laying the foundation of a comprehensive framework for molecular programming. Our language addresses the key challenge of embedding familiar imperative constructs into a set of chemical reactions happening simultaneously and manipulating real-valued concentrations. Although some deviation from ideal output value cannot be avoided, we develop methods to minimize the error, and implement error analysis tools. We demonstrate the feasibility of using *CRN*++ on a suite of well-known algorithms for discrete and real-valued computation. *CRN*++ can be easily extended to support new commands or chemical reaction implementations, and thus provides a foundation for developing more robust and practical molecular programs.

1 Introduction

A highly desired goal of synthetic biology is realizing a programmable chemical controller that can operate in molecular contexts incompatible with traditional electronics. In the same way that programming electronic computers is more convenient at a higher level of abstraction than that of individual flip-flops and logic circuits, we similarly expect molecular computation to admit specification via programming languages sufficiently abstracted from the hardware. This paper focuses on developing a compiler for a natural imperative programming language to a deterministic (mass-action) chemical reaction network implementing the desired algorithm. We do not directly make assumptions on how the resulting reactions would be implemented in chemistry. This could in principle be achieved by DNA strand displacement cascades [15], or other programmable chemical technologies such as the PEN toolbox [3].

Deterministic (mass-action) chemical kinetics is Turing universal [9], thus in principle allowing the implementation of arbitrary programs in chemistry. Turing universality was demonstrated by showing that arbitrary computation can be embedded in a class of polynomial ODEs [4], and then implementing these polynomial ODEs with mass-action chemical kinetics. While these results establish a sound theoretical foundation and show the power of chemistry for handling computation tasks in general, translating and performing specific computational tasks can lead to infeasibly large and complex sets of chemical reactions.

In this work we develop a programming paradigm for chemistry, based on the familiar imperative programming languages, with the aim of making molecular programming more intuitive, and efficient. Most commonly used programming languages such as C, Java and Python, are imperative in that they use statements that change a program's state, with typical branching constructs such as if/else, loops, etc. Note that although CRNs are sometimes talked about as a programming language [7], they are difficult to program directly (it is even unfair to equate them with assembly language). In contrast, *CRN++* operates at a much higher level.

A mapping of imperative program logic to chemical reactions manipulating continuous concentrations poses various challenges that we must address. All reactions happen concurrently, making it difficult to represent sequential computation where, for example, the result of one operation is first computed and then used in another operation. Similarly, all branches of the program execution (i.e., if / else) are followed simultaneously to some degree.

We introduce the syntax and semantics of *CRN++*, which is, to our knowledge, the first imperative programming language which compiles to deterministic (mass-action) chemical reaction networks. *CRN++* has an extensible toolset including error analysis, as well as simulation framework [8]. We thus provide an automatic environment for simulating experiments based on *CRN++* programs.

A user specifies a *sequence* of statements, termed commands, to execute. Assignment, comparison, loops, conditional execution, and arithmetic operations are supported. The generated reactions are logically grouped into modules performing the corresponding command. Each module transforms initial species concentrations to their steady-state values which are the output of the module. We ensure that such modules are composable by preserving the input concentrations at the steady-state. Note that in mass-action chemistry all species occur with non-zero concentrations, and thus all reactions happen in parallel to some extent. To mimic sequential execution, we ensure that the reaction corresponding to the current command happens quickly, while other reactions are slow. For this we rely on a chemical oscillator in which the *clock* species oscillate between low and high concentrations, and sequential execution is achieved by catalyzing reactions with different clock species. To achieve conditional execution, we further need to ensure that the reactions corresponding to the correct execution branch readily occur, while those corresponding to other branches are inhibited. Our *cmp* module sets *flag* species to reflect the result of comparison, and these species catalyze the correct branch reactions.

Sequential execution as well as conditional branching leads to errors. Error is present because instructions (reactions) that should not execute, still do (at a smaller rate, of course). Moreover, the set of basic modules, such as addition, converge to the correct value only in the limit, thus computing approximately in finite time. To mitigate the error, we choose the set of modules to exhibit exponential (fast) convergence, and we provide a toolkit for error analysis and detection. Our tool quantifies the error, which can help a user identify the source of error, and guide the design of more accurate *CRN++* programs.

We demonstrate the expressiveness of our language by implementing and simulating common discrete algorithms such as greatest common divisor, integer division, finding integer square root, as well as real-valued (analog) algorithms such as computing Euler's number and computing π. We implement the *CRN++* compiler to reactions in Mathematica, and use the *CRNSimulator* package [8] to manipulate and simulate chemical reactions. *CRN++* is an extensible programming language allowing for easy addition of new modules; we release the open-source version [1] of the tool to enable others make use of it, and extend it further.

2 Examples

In this section we discuss the characteristics of chemical reaction networks (CRNs) through examples. First, the overall idea of computation in CRNs is presented, followed by example programs in *CRN++*. The focus is to give a high level idea of our technique, while later sections discuss internal details.

Although historically the focus of the study of CRNs was on understanding the behavior of naturally occurring biological reaction networks, recent advancements in DNA synthesis coupled with general methods for realizing arbitrary CRNs with DNA strand displacement cascades [15] opened the path to engineering with chemical reactions. In this work we are not interested in a way to engineer the molecules implementing a reaction but focus on reaction behavior and dynamics. We abstract away molecule implementation information and denote molecular species with letters (e.g. A).

Molecular systems exhibit complex behaviors governed by chemical reactions. To give a formal notation of chemical reaction networks, consider the CRN 1:

CRN 1 Example chemical reaction network

$$A + B \xrightarrow{1} A + B + C \tag{1}$$

$$C \xrightarrow{1} \emptyset \tag{2}$$

The CRN 1 consists of two reactions. A chemical reaction is defined with *reactants* (left side), *products* (right side), and *rate constant* which quantifies the rate at which reactants interact to produce products. To illustrate this, reaction 1 is composed of $reactants = \{A, B\}$, $products = \{A, B, C\}$, and rate constant $k = 1$. Since most reactions in *CRN++* have the rate constant equal to 1, from now on we drop the rate constant when writing reactions, unless it is different than 1. Note that multiple molecules of same species can be in a list of reactants

(analogously for products); to support this we use the multiset notation. As an example, to describe reaction: $A + A \longrightarrow B$ we write $reactants = \{A^2\}$, where the upper index (2) represents multiplicity (number of occurrences).

It may seem that a molecule of C is produced out of nothing in reaction 1, since the multiset of reactants is a submultiset of the products. This represents a level of abstraction where *fuel* species that drive the reaction are abstracted away (i.e., the first reaction corresponds to $F + A + B \longrightarrow A + B + C$). Making this assumption allows us to focus on the computationally relevant species. The choice to use general (non-mass/energy preserving) CRNs is an established convention for DNA strand displacement cascades [15].

When the molecular counts of all species are large, and the solution is "well-mixed", the dynamics of the system can be described by ordinary differential equations (mass-action kinetics). Molecular concentrations are quantified by a system of ODEs, where concentration of each species is characterized by an ODE:

$$\frac{d[S]}{dt} = \sum_{\forall rxn \in CRN} k(rxn) \cdot netChange(S, rxn) \cdot \prod_{\forall R \in reactants(rxn)} [R]^{m_{rxn}(R)}(t)$$

This ODE characterizes concentration of species S ($[S]$), in a given CRN. The right side is a sum over reactions in the CRN, where $k(rxn)$ is a rate of reaction rxn, and $netChange(S, rxn)$ is a net change of molecules of S upon triggering of rxn (can be negative). Concentration of a reactant R in time is written $[R](t)$, while $m_{rxn}(R)$ is the multiplicity of reactant R in reaction rxn. To illustrate the general formula, the set of ODEs characterizing CRN 1 is:

$$\frac{d[A]}{dt} = 0, \frac{d[B]}{dt} = 0, \frac{d[C]}{dt} = [A](t) \cdot [B](t) - [C](t)$$

The $[A]$ and $[B]$ are constant (derivatives zero); thus $\frac{d[C]}{dt} = [A](0) \cdot [B](0) - [C](t)$. From this equality follows that $[C](t)$ is increasing when smaller than $[A](0) \cdot [B](0)$, decreasing in the opposite case, and does not change when $[C](t) = [A](0) \cdot [B](0)$. Thus this system has a global stable steady-state $[C] = [A](0) \cdot [B](0)$. We say that this module computes multiplication, due to the relation between initial concentrations and concentrations at the steady state.

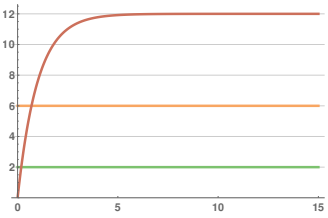


Fig. 1: Multiplication CRN. $[A]$ shown in orange, $[B]$ in green, and $[C]$ in red.

We simulate and plot the dynamics of the multiplication CRN, as shown in Figure 1. Initial concentrations of A and B are 6 and 2, respectively, while the concentration of C approaches value 12. Note that the exact value defined by the steady state ($[C](t) = 12$) is reached only at the limit of time going to infinity. Since the computation has to be done in finite time, the presence of error is unavoidable. This error raises challenging issues with programming in chemistry, and necessitates techniques for controlling it. One crucial property that determines the error is the convergence speed of the

module. The multiplication command in *CRN++* is implemented through the above module, following the design principles of *convergence speed* and *composability* described in section 3. Chemical reactions are abstracted away from a user who can simply write $mul[a, b, c]$ to multiply.

CRN++ is an imperative language, and as such supports sequential execution. Note that even a simple operation of multiplying and storing into the same variable, e.g. $A := A * B$, requires support for sequential execution. We use operator ":=" to relate input and output concentrations; $A := A * B$ denotes that $[A](t)$ converges to $[A](0) * [B](0)$. The above implementation of the *mul* module necessarily assumes that the output species is different from the input species. Otherwise, $mul[a, b, a]$ goes to infinity or 0 depending on the value of B. To implement $A := A*B$, we split the computation into two sequential steps: (1) $C := A*B$, (2) $A := C$. To multiply we use the *mul* module described above. For the assignment we use the load module (*ld*). To ensure the assignment executes after the multiplication, we catalyze the two modules with the clock species that reach their high values in different phases of the oscillator. Importantly, the chemical oscillator and clock species are abstracted away from a user, who simply uses the *step* construct to order reactions: *step*[{*mul*[a,b,c]}], *step*[{*ld*[c,a]}].

One of the basic blocks of programming languages are conditional branches, executing upon success of a precondition. Similarly to implementing sequential operations, we implement conditional execution by activating (through catalysis) some reactions and deactivating others, depending on a result of condition. Since no species can be driven to 0 in finite time[1], all branches of condition will be active to some extent, which makes this an interesting source of errors without direct analogy in digital electronics. In contrast to sequential computation catalyzed by clock species, conditional blocks are catalyzed by *flag* species. The flag species have high and low values that reflect the result of the comparison. Our *cmp* module sets the flag species to reflect the result of the comparison. In the following example we demonstrate the usage of *cmp* module and conditional execution.

```
1: procedure GCD(a, b)
2:     while a ≠ b do
3:         if a > b then
4:             a ← a − b
5:         else
6:             b ← b − a
7:         end if
8:     end while
9:     return a
10: end procedure
```

Fig. 2: Euclid's algorithm for computing GCD.

To demonstrate the expressiveness of our language we showcase the implementation of Euclid's algorithm (Figure 2) to compute the greatest common divisor (GCD) of a two numbers. The GCD is computed by subtracting the smaller of the values from the larger one until they become equal.

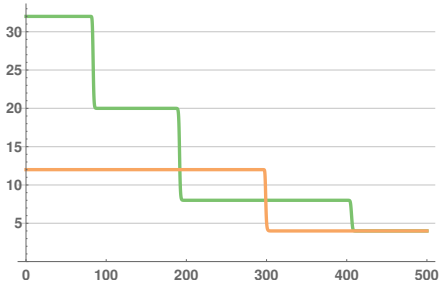
Figure 3a shows the implementation of Euclid's algorithm in *CRN++*. Lines 2-3 define the initial concentrations of species a, b; $a0$ and $b0$ represent the values

[1] Although certain pathological CRNs can drive concentrations to infinity in finite time (e.g., $2A \rightarrow 3A$), and thereby drive certain other species to 0 in finite time (e.g., with an additional $B + A \rightarrow A$), these cases cannot be implemented with any reasonable chemistry.

for which GCD is computed. To order the execution, the *step* construct is used. Multiple instructions that do not conflict with each other can be part of the same step and they are executed in parallel. In the first step a and b are stored into temporary variables and compared, setting the flag species to reflect the result of the comparison. The second step uses the result of the previous comparison, and effectively stores $a - b$ into a if $a > b$, and vice versa. Since the same species cannot be used as both input and output to *sub* module, temporary variables are used (*atmp* and *btmp*). Steps repeatedly execute due to the oscillatory behavior of the clock species, thus implementing looping behavior by default; the steps can be viewed as being inside of the 'forever' loop. *CRN++*, in addition to the language and compiler to chemical reactions, is connected to the simulation backend that enables convenient testing for correctness. We show simulation of the GCD program in Figure 3b where GCD(32,12) is computed. Steps repeatedly trigger causing a and b to converge to the correct result after a couple iterations.

```
1 crn = {
2   conc[a,a0],
3   conc[b,b0],
4   step[{
5     ld[a, atmp],
6     ld[b, btmp],
7     cmp[a,b]
8   }],
9   step[{
10    ifGT[{ sub[atmp,btmp,a] }],
11    ifLT[{ sub[btmp,atmp,b] }]
12  }]
13 };
```

(a) GCD implementation



(b) Dynamic simulation of the GCD program for $a0 = 32$, $b0 = 12$. Concentrations of a (green), and b (orange) are shown in function of time.

Fig. 3: Implementation of Euclid's algorithm for computing GCD in *CRN++* (left), simulation results of the implementation (right).

In addition, we implement a set of algorithms in (a) discrete space—counter, integer division, integer square root, as well as in (b) continuous space, by implementing *CRN++* programs that approximate value of *Euler's* constant and π. These examples are shown in Appendix B.

3 Technique

This section explains *CRN++*, both the underlying constructs used to build it, as well as high level primitives that represent the language itself. We start by presenting high-level modules that are at the core of *CRN++* (section 3.1), followed by explanation of how the sequential behavior is achieved (section 3.2), after which we give an overview of *CRN++* grammar (section 3.3), and finally we discuss the error detection and analysis tools we provide (section 3.4).

3.1 Modules

Modules represent the core of *CRN++*, and in their form are somewhat analogous to the instruction set architecture (ISA) in machine languages. Modules implement basic operations such as load, add, subtract, multiply, compare. We provide the exhaustive list of modules in Appendix A. Importantly, *CRN++* is extensible, and supports easy addition of new modules.

There are multiple ways of computing addition and other operations in chemistry. As mentioned in the previous section, our implementation choice is led by two basic principles: (a) convergence speed, (b) composability.

3.1.1 Convergence speed Consider CRN implementing addition:

CRN 2 Addition CRN (inputs preserved). Inputs: A and B, output: C.

$$A \longrightarrow A + C$$
$$B \longrightarrow B + C$$
$$C \longrightarrow \emptyset$$

By solving the system of ODEs that characterize the concentration of C we get the following equation: $[C](t) = [A] + [B] + ([C](0) - [A] - [B]) \cdot e^{-t}$.

$[C](t)$ is concentration of species C at time t; accordingly $[C](0)$ is initial concentration. Since $[A](t)$ and $[B](t)$ are constant we simply write $[A]$ and $[B]$. From the equation above it follows that $[C]$ converges to the value $[A]+[B]$, and thus we say the CRN performs addition. To consider the convergence speed we look at the non-constant part of the equation. Due to the factor e^{-t} the decrease of the non-constant part is exponential, thus we say that the CRN exhibits *exponential* convergence speed. The convergence speed is of great importance, since it directly affects computation error; the sooner reaction converges the sooner it approaches the correct value.

3.1.2 Composability There are alternative ways to implement addition and have exponential convergence speed:

CRN 3 Addition CRN (destructs inputs). Inputs: A and B, output: C.

$$A \longrightarrow C$$
$$B \longrightarrow C$$

For easier discussion, let us call the initial addition module $CRNAdd_1$ (CRN 2), and the one above $CRNAdd_2$ (CRN 3). To compute $E := (A*B)+D$ we combine the *mul* module (CRN 1), computing $C := A*B$, with an addition module, computing $E := C + D$. If $CRNAdd_1$ is used, multiplication converges to the correct value, after which $CRNAdd_1$ has correct values at its inputs and converges to the correct value of E. Before the multiplication converges and C becomes equal to $A*B$, reactions of $CRNAdd_1$ trigger, but since the module is input-preserving they do not affect steady state of the multiplication module. However, $CRNAdd_2$ consumes its inputs, and the composition will give an incorrect result. The *mul*

CRN constantly drives C to value $A * B$, and will keep refilling inputs to the $CRNAdd_2$, causing the wrong result. This is the reason $CRNAdd_1$ is preferred over $CRNAdd_2$. Moreover, the composed CRN of the *mul* module and $CRNAdd_1$ exhibits the exponential convergence speed, and has a unique stable steady state, the formal proof can be found in work by Buisman et al. [5].

We have set up the two main design criteria (convergence speed and composability) for the modules, and we next describe the core modules of *CRN++*.

3.1.3 *Ld* Module Loads the value from source (first argument) into a destination (second argument). The CRN used for load operation is following:

CRN 4 Load CRN

$$A \longrightarrow A + B$$
$$B \longrightarrow \emptyset$$

A is the input and B is the output species. This module, similar to *add*, has exponential convergence speed [5]. In addition, the concentration of input species is constant, thus ensuring composability.

3.1.4 *Add* Module Adds two values (first and second argument) and stores the result into destination (third argument). The Add CRN is shown in CRN 2; its convergence speed and composability are already discussed.

3.1.5 *Sub* Module Subtracts the second input value from the first and stores into the destination (third argument).

CRN 5 Subtraction CRN

$$A \longrightarrow A + C$$
$$B \longrightarrow B + H$$
$$C \longrightarrow \emptyset$$
$$C + H \longrightarrow \emptyset$$

The above CRN was generated via evolutionary algorithms [5]; by analyzing its system of ODEs, the network computes subtraction. Input species A and B are not affected and the property of composability is satisfied. Neither we nor Buisman et al. found the analytical solution; however, our simulation results show that the module converges exponentially quickly unless $A = B$ (see the Alternative Design subsection of the *cmp* module below for an analogous, easy to analyze case). In a case inputs, A and B, are close to each other the computation error is higher. The error evaluation tools (section 3.4) help in detecting and analyzing problematic cases (e.g., where A and B are close), thus enabling a user to redesign the CRN. In our examples, A and B usually differ by at least 1. Runtime assertions in the simulation package that automatically notify a user about these kind of problems would help identify the source of the error. Note that many algorithms can be refactored to reduce the error (see section 5).

3.1.6 *Mul* Module Multiplies inputs (first and second argument) and stores into destination (third argument). The multiplication CRN is shown in section 2. This CRN does not affect inputs and has exponential convergence speed [5].

We have presented modules for performing arithmetic operations (*ld*, *add*, *sub*, *mul*). These modules are implemented within a single *step*. Multiple modules can be executed in parallel within a single *step* as long as there is no cyclic dependence between species: for example *mul*[a,b,c] and *add*[c,d,a] forms a cycle, the output of the *mul* is input to the *add*, and vice versa. Also, the CRN implementation imposes the restriction that same species cannot be used as both input and output to the same module. We now introduce the *cmp* module providing for conditional execution, which is executed in two *steps*.

3.1.7 *Cmp* Module Compares the two values, and produces signals (flag species) informing which value is greater or if they are equal.

Alternative Designs. Before explaining our implementation of comparison we discuss alternative implementations, and point out design decisions that lead to the current implementation. One of more obvious ways to implement comparison is using the reaction: $A + B \longrightarrow \emptyset$.

If initially $[A] > [B]$, than at the equilibrium all molecules of B interact with A leaving only A; case $[B] > [A]$ is analogous. The proposed reaction can be used for conditional execution, by using A and B catalytically in reactions that should execute when $[A] > [B]$ and $[B] > [A]$, respectively. The comparison reaction should execute before conditional execution is plausible, thus the comparison is done in a step before the conditionally executed reactions.

The comparison module proposed above does not preserve inputs, and thus it is not composable. This imposes the restriction that in the step in which comparison is used no other module uses the compared values. Our *cmp* module does not have this restriction.

We analyze the ODE describing this CRN to evaluate the convergence speed. Since the amount of B decreases with the same speed as A, we can express $[B](t) = [A](t) + D_0$, where $D_0 = [B](0) - [A](0)$. The following holds:

$$\frac{d[A]}{dt} = -[A](t) * ([A](t) + D_0) \implies [A](t) = \frac{[A](0)D_0}{-[A](0) + [A](0)e^{D_0 t} + D_0 e^{D_0 t}}$$

If $D_0 > 0$ ($[B](0) > [A](0)$) terms with exponential factors tend to infinity, and $[A]$ to zero. Conversely, when $D_0 < 0$, exponential factors converge to zero, and $[A]$ to $-D_0$. A converges exponentially, unless A and B are equal at the beginning ($D_0 = 0$); then the dynamics of A is described with: $[A](t) = \frac{[A](0)}{1+[A](0)t}$

In conclusion, the module converges fast (exponential speed) when operands are different, while the module converges slow (linear speed) when operands are equal (or close to each other). The linear convergence speed is yet another problem that lead to sub-optimal performance of this module. Recall that the comparison module drives the flag species which then catalyze branches that should execute, thus having a chained effect. It is of great importance to have a reliable comparison module.

Lastly, to detect equality with the above proposed module, absence of a species needs to be detected, since both values are driven to zero in a case of equality. Detecting the absence of species in chemistry is itself non-trivial and

error-prone. There are several approaches based on so-called *absence indicators*. Generally speaking, the absence indicator for A is produced at a constant rate and gets degraded by A. The absence indicator has to be produced slowly, or else it will be present in non-negligible concentration even if A is present. The absence indicators in the literature rely on a difference between rate constants of several orders of magnitude. The relatively slow dynamics of the production of the absence indicator lead to a fair amount of error affecting the computation, and necessitate slowing down the clock (i.e., the whole computation) to work properly.

Our Design. The *cmp* module is implemented using two sequentially executed sets of reactions, which trigger in consecutive clock phases. In the first phase, the inputs (X and Y) are mapped to flag species $XGTY$ and $XLTY$. Values are mapped to the range [0-1], by setting the initial concentrations of $XGTY$ and $XLTY$ to 0.5. If, for example, $[X] = 80$ and $[Y] = 20$, signal species $XGTY$ and $XLTY$ converge to 0.8 and 0.2, respectively. The mapping is done in order to preserve original values of the inputs, since the next phase of comparison consumes the compared values (flags), thus mapping allows the inputs to be used freely in other instructions. The mapping CRN is shown in CRN 6, and exhibits exponential convergence speed according to our analysis.

CRN 6 CRN for mapping compared values

$$XGTY + Y \longrightarrow XLTY + Y$$
$$XLTY + X \longrightarrow XGTY + X$$

The goal of the second phase of comparison is to detect which value is greater. We use a chemical *Approximate Majority* (AM) algorithm [6] to detect if $XGTY$ or $XLTY$ is in the majority. All molecules of the less populous species convert to the more populous species. AM reactions are:

CRN 7 Approximate Majority CRN

$$XGTY + XLTY \longrightarrow XLTY + B$$
$$B + XLTY \longrightarrow XLTY + XLTY$$
$$XLTY + XGTY \longrightarrow XGTY + B$$
$$B + XGTY \longrightarrow XGTY + XGTY$$

The majority algorithm causes convergence of $XGTY$ to 1 and $XLTY$ to 0 when $X > Y$, and vice versa. The species $XGTY$ are used as a catalysts in reactions that execute when $X > Y$, and the species $XLTY$ for the opposite case. The AM network has been studied in the stochastic context (stochastic CRNs) and is known to converge quickly, even when inputs are close [2].

Equality checking. Due to the always present error in chemical computation, checking for equality is actually approximate-equality checking. Consider having a chemical program with real values, then if the values are close to each other it is impossible to tell if they are actually equal but affected with error, or they represent different real valued signals. Due to this issue, while comparing for equality is impossible, we compare for ϵ-range equality, where ϵ can be

arbitrarily small. For discrete algorithms we use equality checking with $\epsilon = 0.5$, allowing easy comparison of the integer values (e.g., values in range $(2.5, 3.5)$ are considered to be equal to 2). To support equality checking we compare $x + \epsilon$ with y (generating signals $XGTY$ and $XLTY$), and at the same time compare $y + \epsilon$ with x (generating signals $YGTX$ and $YLTX$). Combining the signals of the two comparisons gives the desired result: If $X = Y$, signal $XGTY$ is high ($XLTY$ low) and $YGTX$ is high ($YLTX$ low) due to the added offset. To execute a reaction upon equality both $XGTY$ and $YGTX$ are used catalytically. If $X > Y$, signal $XGTY$ is high ($XLTY$ low) and $YLTX$ is high ($YGTX$ low), so both $XGTY$ and $YLTX$ should be used catalytically. Symmetrically for $X < Y$, both $XLTY$ and $YGTX$ are used catalytically. Note that unlike in the previously proposed comparison module, this module does not use absence checks and absence indicators, and as such is more reliable in time-constrained environment. After calling *cmp* in a step, programmer can use *ifGT*, *ifGE*, *ifEQ*, *ifLT*, *ifLE* in subsequent steps to conditionally execute reactions. Note that the flags are active until the next call to the *cmp* module.

3.2 Sequential Execution

CRN++ allows programming in a sequential manner, despite the intrinsically parallel nature of CRNs. To model sequential execution in CRNs there is a need to isolate two reactions from co-occurring, and control the order in which they happen. The key construct we rely on to achieve these goals is a chemical oscillator.

A chemical oscillator is a CRN in which the concentrations of species oscillate between low and high values. The oscillatory CRN [12] we use is described with a following set of reactions:

CRN 8 Oscillator CRN

$$i = 1, ..., n-1 : X_i + X_{i+1} \longrightarrow 2\,X_{i+1}$$
$$X_n + X_1 \longrightarrow 2\,X_1$$

Concentrations of the clock species (X_i) oscillate (see Figure 4). Different clock species have different oscillation phase and reach minimum and maximum at different times. To control the rate at which a reaction fires, clock species are added as both reactant and product (catalyst), in that way preventing reactions from co-occuring and ordering them (see CRN 9). While overlap between the clock species exists, it is small and thus enables sequential execution. To ensure the small overlap, in *CRN++* we use every third clock species, i.e. X_3, X_6, X_9, etc., to catalyze the reactions that execute at different time moments.

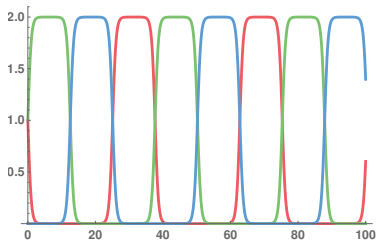


Fig. 4: Chemical oscillator containing 3 species: X_1 (red), X_2 (green), and X_3 (blue).

CRN 9 Ordering reactions: original reactions (left), ordered (right).

$A \longrightarrow B$	$A + X_3 \longrightarrow B + X_3$
$B \longrightarrow C$	$B + X_6 \longrightarrow C + X_6$

The chemical oscillator is abstracted from a *CRN++* user, who can order reactions using the *step* construct. Reactions in different steps are isolated from each other through clock species acting catalytically.

Non-conflicting instructions can be part of the same step. Splitting instructions across multiple steps is needed in a case of a) conditional execution—comparison needs to be done before conditional execution is possible; b) reading and writing to a same species—this is not possible within the step (as discussed earlier), and requires temporal ordering. The number of clock species used is determined by the number of `step` instructions. Each step requires three clock species, with the exception of steps in which *cmp* module is used, for which six clock species are used. The oscillatory behavior of the clock species causes steps to get repeated eventually, causing the loop-like behavior.

3.3 Grammar

Listing 1.1 is an overview of the *CRN++* grammar. At its root the CRN contains a list of *RootS*s, where *RootS* can be either *ConcS* – defines initial concentration of species, *RxnS* – defines a reaction, *ArithmeticS* – performs arithmetic operation, and *StepS* – orders execution. Furthermore, *StepS* is divided into a list of *NestedS*s, where each *NestedS* is either *RxnS*, *ArithmeticS*, *CmpS* – performs comparison, or *ConditionalS*. *ConditionalS* conditionally executes a block based on the result of a previous comparison. Note that the comparison should be done in a step prior to the conditional execution. Based on the result of the comparison, if the first operand is *greater than*, *greater or equal*, *equal*, *less or equal*, *less than* the second operand, conditional block *ifGT*, *ifGE*, *ifEQ*, *ifLT*, *ifLE* is executed.

The grammar can be easily extended; e.g., new arithmetic modules can be added to the list of *ArithmeticS* nonterminals. Aslo, the *CRN++* grammar allows for easy addition of *ifAbsent* conditional statements that can be used to implement the asynchronous programs, in that way enabling comparison between the asynchronous and synchronous programming paradigms.

3.4 Error Evaluation

Programming chemistry is inherently error-prone. We identify three specific sources of error in *CRN++*. First, CRNs converge asymptotically—only in the limit is the correct value reached—thus leaving certain amount of error in a finite time. Second, we cannot completely turn off modules which are not supposed to be currently executing, whether they belong to another sequential step, or to another branch of execution. In addition, comparison has to take into account possible error in the compared values.

Our design decisions were based on minimizing the error; however since error cannot be avoided altogether, we provide a toolkit that helps in error analysis and guiding the CRN (program) design. Using the tool, users can, for any species of interest, track the difference between the correct value, and the (simulated) value in chemistry. For example, if operation add[a,b,c] is executed in a step, than $c = a + b$ is expected in the following step. *CRN++* allows measuring the difference between the expected $c = a + b$, and actual simulation value. This helps users analyze the error, and detect if the error builds up over time.

We analyze the value of operand a from GCD example Figure 3, and plot the error in Figure 5. In Figure 5, the x-axis represents time, while the y-axis shows the difference between expected and actual value of a. Note that the error is sufficiently small that the algorithm executes correctly throughout the analyzed time. The error is not constant, which opens interesting questions of correlating the error with instructions in the program. To correlate error with program instructions we examine the GCD simulation (Figure 3b). By looking at the time axis, it is easy to connect the first two spikes of the error with the subtraction of a.

```
⟨Crn⟩ ::= 'crn= {' ⟨RootSList⟩ '}'

⟨RootSList⟩ ::= ⟨RootS⟩
  | ⟨RootS⟩ ',' ⟨RootSList⟩

⟨RootS⟩ ::= ⟨ConcS⟩
  | ⟨RxnS⟩
  | ⟨ArithmeticS⟩
  | ⟨StepS⟩

⟨ConcS⟩ ::= 'conc['⟨species⟩', '⟨number⟩']'

⟨RxnS⟩ ::=
    'rxn['⟨Expr⟩','⟨Expr⟩','⟨number⟩']'

⟨ArithmeticS⟩ ::=
    'ld ['⟨species⟩','⟨species⟩]
  | 'add ['⟨species⟩','⟨species⟩','⟨species⟩]
  | 'sub ['⟨species⟩','⟨species⟩','⟨species⟩]
  | 'mul ['⟨species⟩','⟨species⟩','⟨species⟩]

⟨CmpS⟩ ::= 'cmp ['⟨species⟩','⟨species⟩]

⟨StepS⟩ ::= 'step[' NestedSList']'

⟨NestedSList⟩ ::= ⟨NestedS⟩
  | ⟨NestedS⟩ ',' ⟨NestedSList⟩

⟨NestedS⟩ ::= ⟨RxnS⟩
  | ⟨ArithmeticS⟩
  | ⟨CmpS⟩
  | ⟨ConditionalS⟩

⟨ConditionalS⟩ ::= 'ifGT['⟨NestedSList⟩']'
  | 'ifGE['⟨NestedSList⟩']'
  | 'ifEQ['⟨NestedSList⟩']'
  | 'ifLT['⟨NestedSList⟩']'
  | 'ifLE['⟨NestedSList⟩']'

⟨Expr⟩ ::= ⟨species⟩ { '+' ⟨species⟩ }
```

Listing 1.1: *CRN++* Grammar

We provide the error evaluation framework with the vision of it being a guiding element for programming in *CRN++*. We found this technique particularly useful for validation of programs, analyzing the error, understanding the sources of error, and redesigning the CRN for correctness.

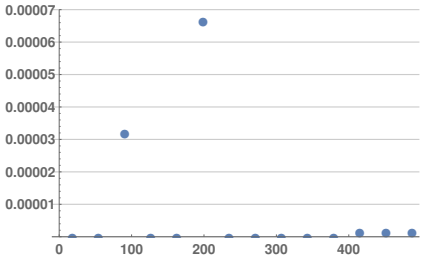


Fig. 5: Error evaluation of species a from GCD program.

4 Related Work

Computational power of chemical reaction networks. Previous research demonstrated techniques of achieving complex behaviors in chemistry, such as: computing algebraic functions [5], polynomials [14], implementing logic gates [13]. Moreover, the Turing completeness of chemistry has been proven, using the strategy of implementing polynomial ODEs (which have been previously shown to

be Turing universal) in mass-action chemical kinetics [9]. Even though Turing complete, this translation to chemistry can result in infeasibly complex chemical reaction networks, which motivates other, more direct methods.

Modular Reactions. Adding even a single reaction to a CRN can completely change its dynamics, which makes the design process challenging. The idea of 'composable' reactions seeks a set of reactions that can be composed in a well-defined manner to implement more complex behaviors. Buisman et al. [5] compute algebraic expressions by designing the core modules that implement basic arithmetic operations, which can be further composed to achieve more complex tasks. Our goal is to make modular designs, and we follow some of the proposed design principles for achieving the goal, such as input-preserving CRNs.

Synchronous computation. Previous work utilized synchronous logic to achieve complex tasks. Soloveichik et al. implement state machines in chemistry by relying on clock species [15]. We use the same technique, where we add clock species acting catalytically to order reactions. Jiang et al.[11], also relying on clock species, design a model of memory in chemistry to support sequential computation, demonstrating their technique on examples of a binary counter and a fast Fourier transform (FFT). Previous work shows the promise of programming synchronous logic in reactions, which we advance by providing an explicit programming language and framework for designing and testing wide-range of programs.

Asynchronous computation. Recall, an absence indicator is a species that is present in high concentration when a target species is present in low concentration. Absence indicators can be used to drive a reaction when a particular reaction has finished, providing a method for executing modules in desired order. Huang et al. [10] use absence indicators to implement algorithms such as integer division and GCD. Their method requires two reaction rates, 'fast' and 'slow', where the fast rate needs to be orders (2-3) of magnitude larger to ensure the proper function of the system. Since, in practice, biochemical systems allow for a restricted range of reaction rates, requiring a large spectrum of rates slows down the computation when the computation speed is dictated by the slow rates. In contrast, we allow all reactions to take the same (or comparable) rate constants. While the goal of our work is not to compare asynchronous and synchronous computation, we mention insights and intuition of their differences, which we gained through empirical studies. First, absence indicators are not robust, and typically require fine tuning to get the system right. Second, error detection is easier with synchronous logic. Since all operations follow the clock signal, there is a direct mapping from a time moment to a command that is executing, which provides a way to check correctness of a system at any point of time. Finally, we provide a framework for implementing molecular programs which is easily extensible, and can be used to compare synchronous and asynchronous logic. We include support for absence indicators through the *ifAbsent* construct, thus allowing easy comparison of the two paradigms.

5 Discussion and conclusions

There are multiple ways in which we can further improve *CRN++*. Note that currently every high-level module is mapped to exactly one CRN implementing the operation. Letting the tool decide which implementation to use in different contexts could boost the performance. For example, the described modules have a useful property of preserving inputs, but that property might not be needed in every case. If the input preserving property is redundant, *CRN++* could choose to use the more optimized version (for example the more compact subtraction CRN discussed above). Also, we could provide a more flexible programming experience by (a) letting the compiler automatically schedule instructions to different steps (instead of the explicit `step` construct); (b) allowing the same species as both input and output of a module and automatically generate the additional instructions.

We plan to further explore the support for nested loops in *CRN++*. Currently nested loops can be mimicked through conditional execution: the loop condition is computed through comparison and the main loop conditionally executes the instructions of the desired loop. Besides explicit support for nested loops, future work will support nested conditionals by adding multiple flag species for multiple comparisons.

An important direction for future research concerns reducing the error in our construction, and understanding how it builds up over time. We noticed that different algorithms, even computing the same function, accumulate varying levels of error. For example, as seen in Appendix C.1, the error of the Sub module increases with the magnitude of the operands, and also increases the closer they are. However, we also found an alternative way to subtract, that keeps the error constant and independent of the operands (see Figure 14b) at the cost of a slower run-time.

Our error analysis shows that for most examples we tried, but not all, error builds up over the course of the computation. For *CRN++* programs where the error builds up in this way, there is some maximum input complexity beyond which the error overwhelms the output. Can all *CRN++* programs be refactored (preferably automatically) to bound the cumulative error of every module such that it does not build up over time? Note that if this were possible, we would obtain another, more efficient, way to achieve Turing universality.

To the best of our knowledge we are the first to provide an imperative programming language which compiles to chemical reaction networks. Moreover, we build tools that can help users get a better understanding of CRNs and improve their design. Although absolutely correct computation is not achieved, we provide tools that help understand why error occurs, and thus help improve the design of CRNs. We release our toolkit as open-source, to encourage new research and improvement of the *CRN++*, with the hope of advancing the engineering of information processing molecular systems.

Acknowledgments. We thank the fellow students of EE 381V (Programming With Molecules) at The University of Texas at Austin for constructive discussions on the material presented in this paper. We also thank Keenan Breik,

Cameron Chalk, Milos Gligoric, Aleksandar Milicevic, Boya Wang and Kaiyuan Wang for their feedback on this work. This research was partially supported by the US National Science Foundation under Grants Nos. CCF-1618895, CCF-1718903, CCF-1652824, and CCF-1704790.

References

1. *CRN++* Github page, `https://github.com/marko-vasic/crnPlusPlus`
2. Angluin, D., Aspnes, J., Eisenstat, D.: A simple population protocol for fast robust approximate majority. Distributed Computing 21(2), 87–102 (2008)
3. Baccouche, A., Montagne, K., Padirac, A., Fujii, T., Rondelez, Y.: Dynamic DNA-toolbox reaction circuits: a walkthrough. Methods 67(2), 234–249 (2014)
4. Bournez, O., Graça, D.S., Pouly, A.: Polynomial time corresponds to solutions of polynomial ordinary differential equations of polynomial length. Journal of the ACM 64(6), 38 (2017)
5. Buisman, H.J., ten Eikelder, H.M.M., Hilbers, P.A.J., Liekens, A.M.L.: Computing algebraic functions with biochemical reaction networks. Artificial Life pp. 5–19 (2009)
6. Cardelli, L., Csikász-Nagy, A.: The cell cycle switch computes approximate majority. Scientific reports 2, 656 (2012)
7. Chen, Y.J., Dalchau, N., Srinivas, N., Phillips, A., Cardelli, L., Soloveichik, D., Seelig, G.: Programmable chemical controllers made from DNA. Nature nanotechnology 8(10), 755 (2013)
8. CRNSimulator Mathematica package, `http://users.ece.utexas.edu/~soloveichik/crnsimulator.html`
9. Fages, F., Le Guludec, G., Bournez, O., Pouly, A.: Strong Turing completeness of continuous chemical reaction networks and compilation of mixed analog-digital programs. In: International Conference on Computational Methods in Systems Biology. pp. 108–127 (2017)
10. Huang, D.A., Jiang, J.H.R., Huang, R.Y., Cheng, C.Y.: Compiling program control flows into biochemical reactions. In: Proceedings of the International Conference on Computer-Aided Design. pp. 361–368 (2012)
11. Jiang, H., Riedel, M., Parhi, K.: Synchronous sequential computation with molecular reactions. In: 2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC). pp. 836–841 (2011)
12. Lachmann, M., Sella, G.: The computationally complete ant colony: Global coordination in a system with no hierarchy. In: European Conference on Artificial Life. pp. 784–800. Springer (1995)
13. Magnasco, M.O.: Chemical kinetics is Turing universal. Physical Review Letters 78(6), 1190 (1997)
14. Salehi, S.A., Parhi, K.K., Riedel, M.D.: Chemical reaction networks for computing polynomials. ACS Synthetic Biology 6(1), 76–83 (2017)
15. Soloveichik, D., Seelig, G., Winfree, E.: DNA as a universal substrate for chemical kinetics. Proceedings of the National Academy of Sciences 107(12), 5393–5398 (2010)

Appendix A Modules

Type	Restrictions	Output (Steady State)	CRN
ld[A,B]	$B \not\equiv A$	$B := A$	$A \longrightarrow A + B$ $B \longrightarrow \emptyset$
add[A,B,C]	$C \not\equiv A \wedge C \not\equiv B$	$C := A + B$	$A \longrightarrow A + C$ $B \longrightarrow B + C$ $C \longrightarrow \emptyset$
sub[A,B,C]	$C \not\equiv A \wedge C \not\equiv B$	$C := \begin{cases} A - B, & A > B \\ 0, & \text{otherwise} \end{cases}$	$A \longrightarrow A + C$ $B \longrightarrow B + H$ $C \longrightarrow \emptyset$ $C + H \longrightarrow \emptyset$
mul[A,B,C]	$C \not\equiv A \wedge C \not\equiv B$	$C := A \cdot B$	$A + B \longrightarrow A + B + C$ $C \longrightarrow \emptyset$
div[A,B,C]	$C \not\equiv A \wedge C \not\equiv B$	$C := A/B$	$A \longrightarrow A + C$ $B + C \longrightarrow B$
$sqrt$[A,B]	$B \not\equiv A$	$B := \sqrt{A}$	$A \xrightarrow{1} A + B$ $B + B \xrightarrow{\frac{1}{2}} \emptyset$
am[A,B]	$A \not\equiv B$	$A := \begin{cases} A + B, & A > B \\ 0, & B > A \end{cases}$ $B := \begin{cases} 0, & A > B \\ A + B, & B > A \end{cases}$	$A + B \longrightarrow A + T$ $B + A \longrightarrow B + T$ $T + A \longrightarrow A + A$ $T + B \longrightarrow B + B$
cmp[A,B]	$A \not\equiv B$	Sets flag species	* Two CRNs (mapping and AM) triggering in two consecutive phases (as discussed in the Technique)

Table 1: *CRN++* Modules. The first column denotes the type of the module. The restrictions column imposes compile-time restrictions for using modules, here $\not\equiv$ is used to mean different species (not values). The output column shows the value of outputs at the steady state. Finally, the CRN column shows chemical reactions implementing the module.

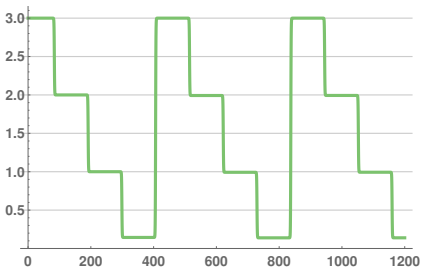
Appendix B *CRN++* Example Programs

Appendix B.1 Discrete Counter

We implement a discrete counter that counts from a predefined value to zero, and repeats the process. Fig 6 shows both *CRN++* program and simulation results. The counter value is stored in the variable c, $cInitial$ preserves the initial value of the counter for later refills, while *one* and *zero* store constants 0 and 1, respectively. The initial concentrations of the species are set up in lines 2-3, note that $c0$ is a parameter representing the initial counter value. The counter is subtracted by one ($cnext := c-1$), and compared with the zero; in the step 1. In a case counter is zero, than its value is reset to the initial value ($c := cInitial$), otherwise it is decremented by one ($c := cnext$), in the step 2. Recall the steps exhibit looping behavior, thus the above process is repetitive.

```
1 crn = {
2   conc[c,c0], conc[ cInitial ,c0],
3   conc[one,1], conc[zero,0],
4   step[{
5     sub[c,one,cnext],
6     cmp[c,zero]
7   }],
8   step[{
9     ifGT[{ ld[cnext,c] }],
10    ifLE[{ ld[ cInitial ,c] }]
11  }]
12 }
```

(a) *CRN++* code.



(b) Simulation results for $c0 = 3$; value of c is shown (green line).

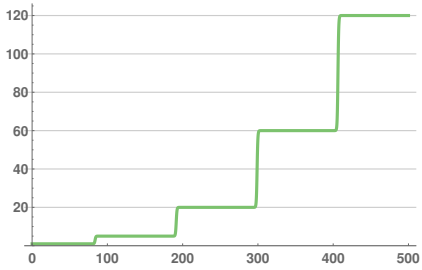
Fig. 6: Discrete Counter.

Appendix B.2 Factorial

We compute the factorial using *CRN++* program. Fig 7 shows both the program and simulation results. To compute the factorial of a number n, we store n in the iterator variable i, and repeatedly multiply f with i, decreasing i until it gets to zero. Initial concentrations of the species are defined in the line 2. In the step 1 (lines 3-7), value of the iterator i is compared with one to check the termination condition; f is multiplied with the i storing the value in the temporary variable $fnext$, and finally the iterator is decremented storing the value in the temporary $inext$. In the step 2 (lines 8-13), commands are executed only $i > 1$, moving the values of temporaries back to f, and i.

```
crn={
  conc[f,1], conc[one,1], conc[i,f0],
  step[{
    cmp[i,one],
    mul[f,i,fnext],
    sub[i,one,inext]
  }],
  step[{
    ifGT[{
      ld[inext,i],
      ld[fnext,f]
    }]
  }]
}
```

(a) *CRN++* code.



(b) Simulation results for $f0 = 5$; value of f is shown (green line).

Fig. 7: Factorial.

Appendix B.3 Integer Division

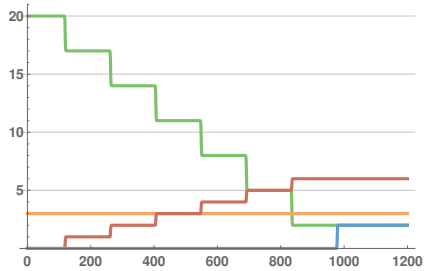
We implement integer division of a two numbers, computing quotient and remainer of the operation. Dividend is stored in the variable a, divisor in b, quotient in q, and remainder in r. Fig 9 shows both program and simulation results. Value of the divisor is subtracted from the dividend, until dividend becomes less than the divisor. In the step 1 (lines 3-5), dividend and divisor are compared to detect if the termination condition is satisfied. In the step 2 (lines 6-11), if $a > b$, the dividend is subtracted by divisor and quotient incremented. In the step 3, if $a > b$, new values for the dividend and quotient are restored from the temporary variables into the original ones. Also, in the step 3, if $a \leq b$, the value of dividend is stored into the remainder (line 17).

1: **procedure** INT SQRT(n)
2: $z \leftarrow 0$
3: **while** $(z+1)^2 \leq n$ **do**
4: $z \leftarrow z + 1$
5: **end while**
6: **return** z
7: **end procedure**

Fig. 8: Integer square root.

```
1 crn={
2   conc[a,a0],  conc[b,b0],  conc[one,1],
3   step[{
4     cmp[a,b]
5   }],
6   step[{
7     IfGE[{
8       sub[a,b,anext],
9       add[q,one,qnext]
10    }]
11  }],
12  step[{
13    IfGE[{
14      ld[anext,a],
15      ld[qnext,q]
16    }],
17    ifLT[{ld[a,r]}]
18  }]
19 };
```

(a) *CRN++* code.



(b) Simulation results for $a0 = 20$, $b0 = 3$; values of a (green), b (orange), q (red), and of r (blue) are shown.

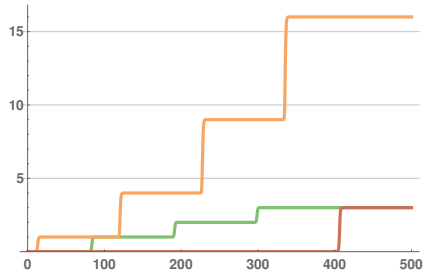
Fig. 9: Division.

Appendix B.4 Integer Square Root

We implement a program that finds integer square root of a number. Figure 8 shows the algorithm; the square root of a number n is found by iterating through numbers $0, 1, 2$, etc, until the power of the iterated number overshoots n. We map the algorithm to *CRN++* program, and show the code and simulation results in Fig. 10. In step 1 (lines 3-7), we increment the z ($znext := z + 1$), compute power of the $z + 1$ ($zpow := znext * znext$), and compare the power with n. In step 2 (lines 8-11), if $zpow < n$, then $znext$ is stored into z, otherwise, the result is computed and stored in the *out*.

```
1 crn = {
2   conc[one,1],  conc[n,n0],
3   step[{
4     add[z,one,znext],
5     mul[znext,znext,zpow],
6     cmp[zpow,n]
7   }],
8   step[{
9     ifLT[{ld[znext,z]}],
10    ifGE[{ld[z,out]}]
11  }]
12 };
```

(a) *CRN++* code.



(b) Simulation results for $n0 = 10$. Values of z (green), $zpow$ (orange), and *out* (red) are shown.

Fig. 10: Integer square root.

Appendix B.5 Euler's number approximation

So far, we presented discrete algorithms, however chemistry allows for real-valued (analog) computations. For programming with real values we extend *CRN++* with additional module performing division – *div*. *div* module follows same design principles and characteristics as other arithmetic modules we presented, and due to the space constraints we do not analyze its CRN.

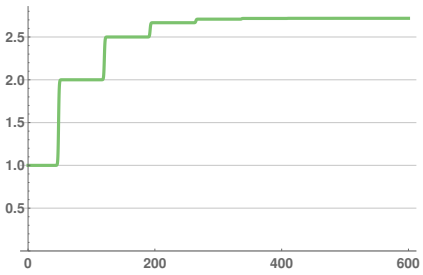
We implement program that approximates Euler's constant. Euler's constant can be computed using the following infinite series:

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} = \frac{1}{1} + \frac{1}{1} + \frac{1}{1 \cdot 2} + \frac{1}{1 \cdot 2 \cdot 3} + ...$$

We map this program into *CRN++* code, as shown in Fig 11. Variable e contains current approximation of the constant, while *element* stores the current element of the series. In step 1 (lines 5-9), the *element* is divded by the *divisor*, *divisor* incremented for the next iteration, and e incremented by the current element of the series. In step 2 (lines 10-14), the temporary variables $elementNext$, $eNext$, and $divisorNext$, are restored into the original ones. Precision achieved at the end of simulation is up to 5 decimal digits, we get result 2.71828.

```
1 crn = {
2   conc[e, 1], conc[element, 1],
3   conc[ divisor , 1], conc[one, 1],
4   conc[ divisorMultiplier , 1],
5   step[{
6     div[element, divisor , elementNext],
7     add[ divisor , one, divisorNext ],
8     add[e, elementNext, eNext]
9   }],
10  step[{
11    ld[elementNext, element],
12    ld[ divisorNext , divisor ],
13    ld[eNext, e]
14  }]
15 };
```

(a) *CRN++* code.



(b) Simulation results; value of e is shown (green line).

Fig. 11: Approximating Euler's constant through infinite series.

Appendix B.6 Approximating π

We approximate the π constant via *CRN++* program. We rely on the following infinite series to do so:

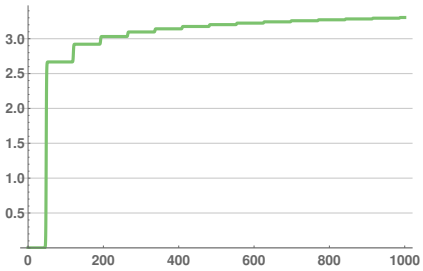
$$\pi = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + ...$$

Fig 12 shows both code and simulation. In step 1 (lines 6-13), 4 is divided by the current divisor $divisor1$ and stored into the $factor1$, also 4 is divided by the

$divisor2 := divisor1 + 2$ and stored into the $factor2$, $factor1$ and $factor2$ are subtracted and added to pi, at the same time $divisor1$ and $divisor2$ are increased by 2 for the next iteration. In step 2 (lines 14-18), the temporary variables $divisor1Next$, $divisor2Next$, and pi are restored to the original variables. Value of pi at the end of simulation is 3.1417. Note that error builds up, if we increase simulation time π converges to value that is in $\epsilon = 0.2$ range of the correct result. This is unlike the approximation of the Euler's constant; error evaluation shows that the reason is due to using the subtraction (of close values) to approximate the π, and subtraction is the most error-prone operation out of all arithmetic modules we present (see Appendix C).

```
1 crn={
2   conc[four , 4],
3   conc[ divisor1 , 1],
4   conc[ divisor2 , 3],
5   conc[pi , 0],
6   step[{
7     div[four , divisor1 , factor1 ],
8     add[ divisor1 , four , divisor1Next ],
9     div[four , divisor2 , factor2 ],
10    add[ divisor2 , four , divisor2Next ],
11    sub[ factor1 , factor2 , factor ],
12    add[pi , factor , piNext]
13  }],
14  step[{
15    ld[ divisor1Next , divisor1 ],
16    ld[ divisor2Next , divisor2 ],
17    ld[piNext, pi]
18  }]
19 };
```

(a) *CRN++* code.



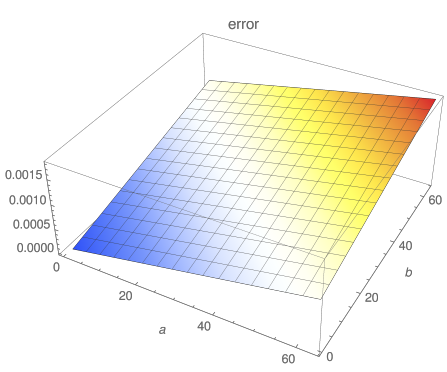
(b) Simulation results; value of pi is shown (green line).

Fig. 12: Approximating Pi constant through infinite series.

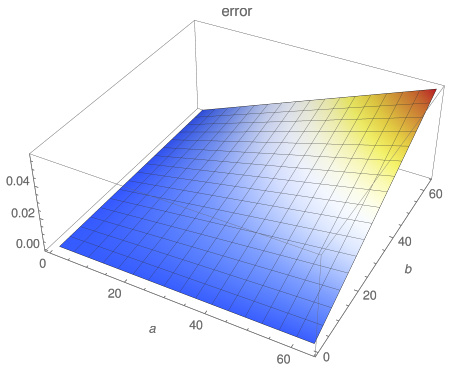
Appendix C Error Characterization

Appendix C.1 Error of Arithmetic Modules

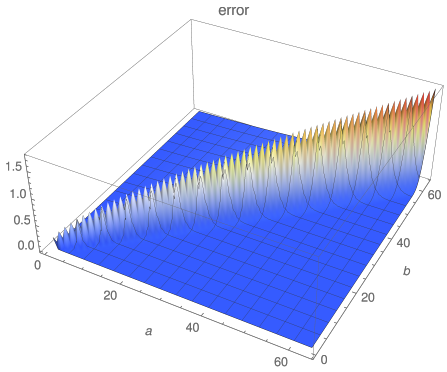
Using our error evaluation mechanisms (section 3.4) we analyze the error of the modules. We evaluate each module separately, on different inputs, to characterize its behavior. Figure 13 shows the error evaluation results, x and y axis reflect values of the first (a) and second (b) operand, respectively, while z axis shows the error. The plots provide useful knowledge: (a) The *mul* module error depends on the value being computed, it increases as the value being computed increases, and does not depend on the order of arguments—preserves commutativity property; (b) The *add* module follows the same pattern as *mul*, but has a lower absolute error; (c) The *sub* exhibits the maximum error when inputs are close to each other, and in general, has higher error rate than other arithmetic modules. This knowledge is useful when designing *CRN++* programs, we realize that the particularly error-prone operation is subtraction of the arguments close to each other; this is indeed the reason why error in example approximating π constant (Appendix B.6) is higher than in the one approximating Euler's number (Appendix B.5). Having this in mind, a user can optimize a program; for example, the subtraction of close operands can be done in alternative, less error-prone way (Figure 14b). We plan to add runtime assertions to *CRN++* programs that alert for possible issues in the program, for example, when values being subtracted are closer than ϵ to each other.



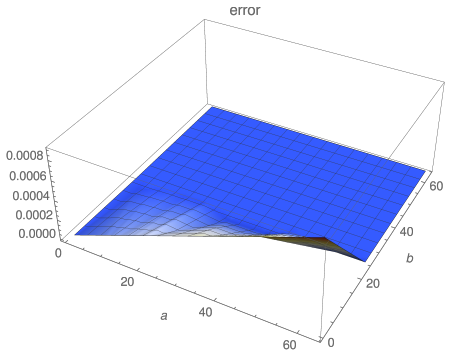
(a) Add evaluation.



(b) Mul evaluation.



(c) Sub evaluation.

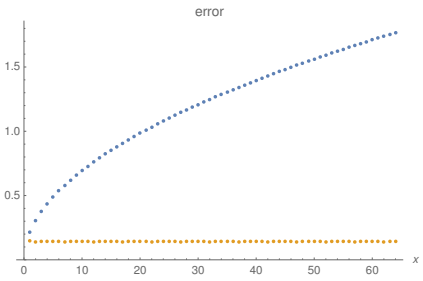


(d) Dvd evaluation.

Fig. 13: Error evaluation of arithmetic modules. Axis a and b show the values of the first and second operand, respectively; z axis show the value of the error (difference between the correct and actual value of the operation).

Appendix C.2 Reducing the error through program refactoring

The *sub* has a high error when operands are close to each other; but there are alternative ways to subtract. Figure 14b shows the alternative code for performing subtraction. Value of b is subtracted from a, by iteratively subtracting 1 from both a and b, until b reaches 0.



(a) Comparing error of Sub module (blue lines) and alternative way to subtract (orange lines). X-axis show the value of both minuend and subtrahend.

```
1 crn = {
2   conc[a, a0], conc[b, b0],
3   conc[one, 1], conc[zero, 0],
4   step[{
5     cmp[b, zero]
6   }],
7   step[{
8     ifGE[{
9       sub[a, one, anext],
10      sub[b, one, bnext]
11    }]
12  }],
13  step[{
14    ifGE[{
15      ld[anext, a],
16      ld[bnext, b]
17    }]
18  }]
19 }
```

(b) Alternative way to subtract.

Fig. 14: Comparing error of *sub* with the alternative way of subtracting (Figure b). Error evaluation is shown (Figure a) for the cases when the operands are equal (minuend and subtrahend same), since *sub* exhibits the highest error in that case.