CRNRepair: Automated Program Repair of Chemical Reaction Networks

Ibrahim Mesecan
Dept. of Computer Science
Iowa State University
Ames, Iowa, USA
imesecan@iastate.edu

Michael C. Gerten
Dept. of Computer Science
Iowa State University
Ames, Iowa, USA
mcgerten@iastate.edu

James I. Lathrop
Dept. of Computer Science
Iowa State University
Ames, Iowa, USA
jil@iastate.edu

Myra B. Cohen
Dept. of Computer Science
Iowa State University
Ames, Iowa, USA
mcohen@iastate.edu

Tomas Haddad Caldas
Polytechnic School
Pontifical Catholic University of Rio Grande do Sul
Porto Alegre, RS, Brazil
tomas.caldas@edu.pucrs.br

***Abstract*—Chemical reaction networks (CRNs) are abstractions of distributed networks that form the foundations of many natural phenomena such as biological processes. These can be encoded and/or compiled into DNA and have been shown to be Turing complete. Before CRNs are implemented in a physical environment, they are often simulated in programming environments. Researchers have recently designed a software testing framework for CRNs, however, repairing CRN programs is still a manual task. Finding and repairing the faults can be difficult without automated support. In this paper we present CRNRepair, a program repair framework for CRN programs. We built our framework on top of an existing APR framework and use a testing infrastructure built in the Matlab SimBiology package. We adapt it to use the SBML representation for its abstract syntax tree. In a case study on 19 mutant versions of 2 programs, we find plausible patches for 90% of one of the programs, and 50% of the other. We find several common types of repairs, which differ from the correct programs, but are functionally correct.**

***Index Terms*—program repair, chemical reaction networks**

I. Introduction

Chemical Reaction Networks (or CRNs) are an emerging paradigm for computing. (see [1], [2] for example) They consist of a set of abstract chemical reactions that represent a distributed computing environment, and have been used for years to describe the interactions between matter [3]. In recent years, technology has evolved to design molecules that can simulate an arbitrary CRN [4]. This transforms the CRN model from a descriptive model of nature to a prescriptive programming language for manipulating matter.

A CRN is characterized by a set of species and a multiset of reactions over the species. Reactions consist of a list of reactants, a list of products, and a reaction rate constant. The species combined with the reactions constitute the program. A common syntactical expression of a CRN is given by

$$X1 \xrightarrow{1} Y$$
$$X2 + Y \xrightarrow{1} null$$

This CRN performs the subtraction of $X1$-$X2$ and places the result in Y (as we will explain in more detail in the next section). There are two reactions with rate constants of 1, and it is common to omit rate constants in this case. In the first reaction $X1$ is the reactant species and Y is the product species. Similarly, in the second reaction $X2$ and Y are the reactant species, and null specifies that there no products (or no product species that we care about).

As in any software programming language, it is easy for programmers to make mistakes. For instance, suppose we mistakenly used $X2$ in the first equation instead of $X1$. This would lead to incorrect behavior. Faults in CRNs can be subtle and are sometimes flaky in behavior [5]. It is also possible to add extra reactions which might appear to create an incorrect program, but instead act as catalysts that speed up the program. We discuss this in our study.

In recent work, we developed a software testing framework, ChemTest [5]. This provides scalability over existing approaches (model checking). While this is an important step in CRN validation, there is no automated approach to localize or repair faults in CRN programs once they are discovered. Even for the small programs we used in our evaluation, it can be difficult to identify which program elements are wrong and to identify a fix. As these programs become more prevalent, automated repair techniques have potential to help developers.

In this paper, we build the first (that we are aware of) automated program repair framework for CRNs called CRNRepair. Figure 1 shows an overview of CRNRepair. It extends an existing genetic improvement framework [6] and combines this with a Matlab SimBiology based testing

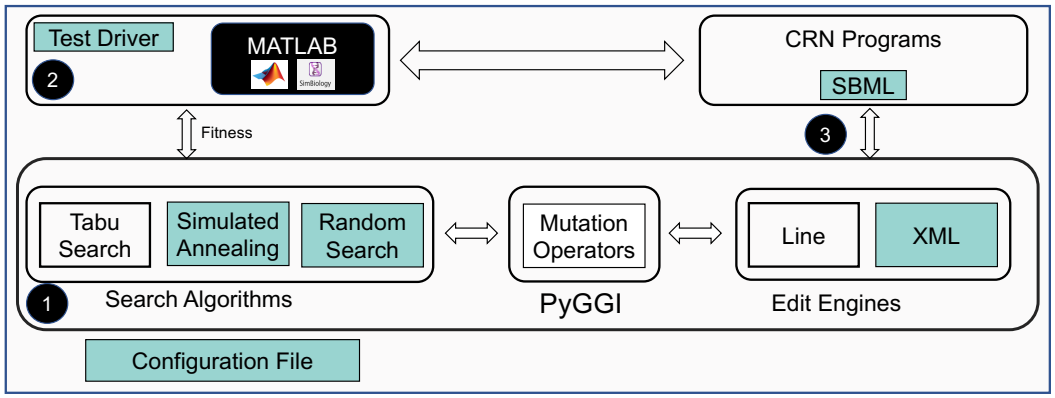

Fig. 1. CRNRepair Framework. Lower portion is the PyGGI engine. Upper portion is the Matlab test driver.

framework. The shaded boxes show our extensions. We added a simulated annealing local search and a random search to the genetic improvement framework. We also extended the tree engine to support the Systems Biology Markup Language (SBML) to represent CRNs [7]. Our test driver module runs SimBiology test cases and returns a fitness based on the test results. Combining these two frameworks together gives us a powerful new tool for the engineering of molecular programs.

The contributions of this paper are:

- A CRN repair framework;
- A case study demonstrating its potential; and
- An evaluation of the types of patches and trade-off between iterations and epochs.

The rest of this paper is organized as follows. Section II gives background information on CRNs and the frameworks and tools used in this paper. Section III describes our algorithms and framework. Sections IV and V presents our case study. We conclude and present future directions in Sections VI and VII.

II. Background

There are two main semantic models or interpretations of the CRN model: deterministic and stochastic, and this paper focuses on the stochastic model [8]. However, it should be noted that this work is easily extended to the deterministic model with minor changes to the software frameworks. The stochastic model operates on individual molecules. We return to the example above to illustrate this model. In the example, initial molecule counts must be assigned to *all* species. For example, $X1$, $X2$, and Y are assigned initial values of 5, 2, and 0, respectively. This example CRN computes the difference of two input values of $X1$ and $X2$ (i.e. $X1 - X2$) with output of 3 in Y when the computation finishes. In this CRN, $X1$ and $X2$ are input species, and Y is the output species.

Since the stochastic model is a distributed computing environment, there are multiple correct execution paths through the CRN program. We illustrate the execution of the subtraction example through a single correct execution path in Table I. Each row is the result of a reaction firing in the computation, and the simulation time that it fires. This time is calculated by Matlab based on physics and the laws of mass action [8]; it is meant to represent real physical time if the chemicals were in nature. Simulation time 0 represents the initial program state, where $X1$ and $X2$ are set to 5 and 2 respectively. Y is initialized to be 0.

TABLE I
Example of a CRN execution

Time	Reaction	$X1$	$X2$	Y
0	Initial Values	5	2	0
0.0410	$X1 \rightarrow Y$	4	2	1
0.0863	$X1 \rightarrow Y$	3	2	2
0.5569	$X2 + Y \rightarrow null$	3	1	1
1.3327	$X1 \rightarrow Y$	2	1	2
1.5840	$X1 \rightarrow Y$	1	1	3
1.6345	$X2 + Y \rightarrow null$	1	0	2
1.6702	$X1 \rightarrow Y$	0	0	3
100	End of Simulation	0	0	3

At simulation time 0 only the reaction $X1 \rightarrow Y$ can *fire* since there are no Y molecules present to enable the reaction $X2 + Y \rightarrow null$. When this first reaction fires (simulation time 0.0410), a molecule of $X1$ reacts and is transformed into a molecule of Y. After this first reaction fires all species have at least one molecule, so it is possible for either reaction to fire. When multiple reactions can fire, the next reaction is chosen randomly according to the law of *mass action* [8]. The order cannot be controlled, hence there are many possible permutations of execution. Suppose the same reaction fires again next (time 0.0863), and another molecule is removed from $X1$ and placed in Y. The rest of the reactions fire to complete a full execution of the CRN which stops at simulation time 1.6702 when no reaction can fire. In this case, the only molecules left are in Y with a value of 3, which is the result of subtraction $(5-2)$. Note that even though the state of $X1$, $X2$, and Y do not change after this time, a simulation of this system may continue to run, in this case until simulation time 100.

A. Testing CRNs

In prior work [5] we designed a testing framework to test the correctness of CRN programs. We used requirements written in a temporal logic and transformed these to abstract test cases which were instantiated with a range of concrete values. The logic provides an oracle which is associated with each abstract test.

Because reactions can fire in different orders, the number of steps (or time of convergence) on a final answer can vary. Hence, a simulation time is chosen to evaluate the answer and if correct at that time, it is considered correct. We also demonstrated that some faults will cause test flakiness (pass sometimes and fail other times), hence the testing framework is run a number of times.

During testing, we differentiate between three different measures of time, CRN computation time, CRN simulation time, and Matlab execution time.

- **CRN Simulation Time.** This is the number simulation seconds that Matlab simulates the CRN network.
- **CRN Computation Time.** This is the minimum number of simulation seconds after which the state of the CRN (the molecule count in each species) no longer changes.
- **Matlab Execution Time.** This is the number of CPU seconds the host computer running Matlab requires in order to run the simulation for *CRN Simulation Time* seconds.

Referring back to Table I we see that the CRN *Simulation Time* is 100 seconds, and that each reaction fires at a time relative to the beginning of the simulation. At time 1.6702 the reaction $X1 \rightarrow Y$ fires, after which there are no $X1$ and $X2$ molecules left. No further reactions can fire, hence the molecule counts for each species does not change, and 1.6702 is the CRN *Computation Time*. Note that the CRN continues to simulate until time 100,but no data is written to the log. Table I is similar to the log files produced by Matlab simulation in that they contain a time, reaction, and molecule counts for each species. Finally, the *Matlab Execution Time* is the real clock time required for Matlab to run the simulation and produce the log output file that contains the trace of the simulation. This metric is important since two distinct behaviors can slow down the execution of the Matlab simulation.

B. GI and Automated program repair

Genetic improvement (GI) [9], [10] and its sub-domain of automated program repair (APR) [11]–[14] is a state of the art technique to evolve program code to improve it in some way. In genetic improvement we usually focus on improving a program for a particular non-functional quality such as energy usage, execution time, memory usage, etc. In APR we focus on correcting faults in a program. There exist many approaches (see [10] and [12] for surveys on GI and APR respectively). Common approaches use stochastic search or evolutionary algorithms. Other approaches use constructive approaches via constraints, templates or program synthesis. To date, APR has been applied primarily to traditional programming languages such as Java, C++ and Python. More recent work has proposed the use of APR for fixing Alloy Models [13].

In this research we leverage the Python General Framework for Genetic Improvement 2.0 (PyGGI) [6] which utilizes a search based approach. It includes algorithms for both program repair and genetic improvement and allows the user to easily plug in different search algorithms and fitness functions. We chose PyGGI because it is built to be extensible, by separating the logic for repair and for interacting with programs and obtaining fitness. More importantly, it can repair programs in multiple programming languages such as Python, Java, C++, and C# by leveraging an XML program representation. It can be extended for any language. The user only needs to provide a parser such as srcML [15] customized for their language and a testing framework. PyGGI uses existing unit testing frameworks such as JUnit or pytest to run test cases and calculate fitness. We describe how we have modified PyGGI to repair CRNs in the next section.

III. CRNRepair

We now present our repair framework CRNRepair, shown in Figure 1. In the center of the figure is the PyGGI framework. This part of CRNRepair controls the search algorithms, the mutation operators and the core patching mechanisms. The shaded boxes indicate places where we have modified PyGGI for CRNRepair. In the top portion, we see a test driver on the left and the CRN program representations on the right. Last, on the bottom left, there is a configuration file which allows us to customize the options for repair. PyGGI provides options such as the number of epochs (the overall trials), number of iterations, and which operators to keep. We keep these, but have added several others. We can change the search algorithm, the timeout delay, and require a directory name within the testing framework (this tells our test driver where to find the unit tests for a particular subject) and allow the user to change the size of the tabu list. We present each modification in more detail next.

A. Search Algorithms (#1)

We begin with our core search algorithms. The PyGGI framework provides a tabu search. The tabu search starts with an empty patch and at each iteration, it randomly selects a mutation operator (replace, insert, delete) and appends it to the existing patch, as long as the resulting patch does not already exist on the tabu list (history of all patches so far). If this patch is better or equal in fitness, it is accepted and this becomes the new patch. While the search uses a tabu list to diversify the search area, it does not allow any patches to be accepted which are worse.

We have added two additional algorithms to the framework. The first is a simulated annealing algorithm. The

second is a random search. We chose simulated annealing because bad moves are allowed with a decreasing probability as the search proceeds [16]. It also takes into consideration the distance from the current (best) patch fitness (see Equations 1-3).

$$Delta = CurrentFitness - NewFitness \quad (1)$$

$$SearchTime = \frac{MaxIterations}{CurrentIteration} \quad (2)$$

$$prob = e^{\frac{Delta}{(2 \times SearchTime)}} \quad (3)$$

In these equations, Delta will be a negative number. The larger it is, and the larger the SearchTime is, the lower the probability of accepting a bad move will be. We keep the tabu list to force the search to move to a broader set of states, but set it to a configurable length, l and after l iterations we drop tabu items in a first in first out fashion. This balances the move to newer states, but does not overly restrict it.

We also implemented a random search which randomly generates a completely new patch of length between 1 and 10 using random mutation operators. In random search each iteration is independent of the existing patch. We heuristically found that most patches were able to find solutions with a relatively short set of edits, hence we restrict random patches to length 10. We examine resulting (successful) patch lengths in our study.

B. Test Driver #2

PyGGI expects a unit test runner to obtain program fitness. For this, we use the Matlab SimBiology application [17]. We call Matlab and pass it the current patched program, run simulations, and return the number of failing simulations. We follow the work of Gerten et al. [5] and run each test case multiple times. We simulate each test case to a fixed simulation time and check our oracle only at that point. For this work we use a simulation time of 100 and use 10 repetitions of each test case. We also set a timeout in the PyGGI program, since some of the mutant programs will not complete 100 iterations in a reasonable amount of Matlab execution time. Our fitness is the number of failed simulations, however, the framework provides additional information that can be returned. We leave other fitness functions as future work.

C. Program Representation #3

One of the advantages of using PyGGI for CRNRepair, is that it provides a choice between line level edits (which work on individual lines of code and are not cognizant of program structure) and tree edits which use an abstract syntax tree (AST) representation of the program. PyGGI uses XML for its tree representation which allows it to represent an arbitrary AST (e.g. it is language independent). To add a new language, one has to provide the parser. For traditional languages such as C++ we can use srcML [15] as a parser to provide the AST.

For CRNs, we can use the Systems Biology Markup Language (SBML) to represent programs [7]. It contains tags for reactions, products, species, rate constants etc. Matlab can directly process SBML, hence it can be passed into our testing framework. In the example (Table I), we used a textual format for our CRNs since these are humanly readable. Internally, we use SBML, but provide the final patched CRNs in both textual and SBML format.

To make the SBML work for CRNRepair, we modified in a few ways. First, since repair operators use only existing variables from the AST and faulty CRNs may exclude some allowable species, we added all species to the SBML. This is done using an annotation and does not impact how Matlab processes the XML. Second, since Matlab requires the Mass Action semantics (we have defaulted to 1.0 for all reactions) and that is not part of the AST tree we have modified the tree parser to handle this.

Last, we added some logic to ensure we are building correct SBML, such as avoiding the removal of both reactants and products at the same time (which leads to a no-op). We note that some CRNs may still not run to completion, however, we avoid many of the obvious compile time issues. As part of this process we have modified the PyGGI xml tree code to work with our programs.

IV. Case Study

In this section, we present a study to evaluate the ability of CRNRepair to create plausible patches in CRNs. We ask two research questions.

RQ1: How effective is CRNRepair? For this research question we measure effectiveness by counting the number of patches obtained for a set of faults and the time to repair. We then examine he different types of patches. We first validate that they are correct repairs, and we then classify the types of repairs we see.

RQ2: What is the impact of differing numbers of epochs and iterations on the repairs? In the second RQ we perform a study to evaluate the impact of varying epochs versus iterations on the repair.

A. Study Subjects

For this work we use two of the subjects from Gerten et al. [5]. We use all of the mutants for two of their programs (`Subtraction` and `Hailstone`). Table II shows details of the two subjects. Subtraction is the same program in our working example previously described. The original program has 2 reactions. Hailstone is a mathematical function that returns $X/2$ when X is even and $3X+1$ when X is odd. It has 11 reactions. The third subject from that paper is probabilistic, therefore we drop it.

In that work there are 9 mutant programs for subtraction and 10 for hailstone. Each program has a single random mutation over the original. These include adding/deleting reactions, adding, removing and/or replacing products or reactants. The mutations for each mutant (S# for subtraction and H# for hailstone) are

TABLE II
SUBJECTS FOR STUDY, WITH EACH REACTION LISTED ON THE RIGHT. MUTANTS BY SUBJECT ON THE LEFT. FOR EACH SUBJECT THE CHANGED REACTION (R) AND REACTION NUMBER (#) IS GIVEN.

Subtraction (S)	ID	Change	R#	Hailstone (H)	ID	Change	R#
$X1 \to Y$	$S1$	$X1 + X1 \to X2$	3	$X1 \to PO + H + M$	$H1$	$2B + 3A \to null$	7
$X2 + Y \to null$	$S3$	$X1 + Y \to null$	2	$PO + PO \to PE$	$H2$	$M \to H$	12
	$S4$	$X1 + X2 \to Y \to null$	3	$PE + PO \to PO$	$H3$	$CE \to 3B + 6A$	6
	$S5$	Removed	1	$PE + PE \to PE$	$H4$	$CE + PO + y \to null$	12
	$S6$	$X2 + Y + X1 \to null$	2	$H + H \to D$	$H5$	Removed	4
	$S7$	$Y \to null$	2	$M \to 3B + 6A$	$H6$	$X1 \to PO + H$	1
	$S8$	$null \to X2 + X1 + Y$	3	$2B + 2A \to null$	$H7$	$CO + PE + Y \to A$	11
	$S9$	$Y \to X2 + X2$	3	$PE + D \to PE + CE + Y$	$H8$	$PO + A + Y \to PO + CO + Y$	9
	$S10$	$X2 \to null$	2	$PO + A \to PO + CO + Y$	$H9$	$2B + 2A + PO \to null$	7
				$CE + PO + Y \to PO + D$	$H10$	$CO + PE + Y + CE \to PE + A$	11
				$CO + PE + Y \to PE + A$			

shown in Table II. They also provide a set of test cases which find the mutants. We utilize these tests in our implementation of CRNRepair. We added additional tests for the validation to avoid overfitting and to ensure our patches are plausible. We discuss this more in RQ1.

B. Methodology

For RQ1 we use the following experimental parameters. We use the simulated annealing algorithm with a rolling tabu list of 15. We use 10 epochs (trials) and 100 iterations. We run experiments on a cluster of high performance computers. Each node has an Intel(R) Xeon(R) Gold 6244 CPU @ 3.60GHz, RedHat Enterprise Linux 7, Matlab 2019a, and 2 GB of RAM.

To check plausibility we extend the test suites with an additional set of manually designed test cases. This adds 16 and 12 concrete tests respectively to the subtraction and hailstone. We run the programs using the set of the original tests plus the new tests. We then manually examine each patch and classify these into categories (1) equivalent to the original, correct program, (2) functionally equivalent programs that speedup the computation, and (3) functionally equivalent programs that slow down the simulation.

For RQ2 we first select a subset of the mutants. We chose 3 subtraction subjects which were harder to find and 4 of the hailstone subjects. We then run simulated annealing using different combinations of epochs and iterations. We doubled the overall iteration budget to 2000 (total epochs and iterations) and then selected sizes (i) 10-200, (ii) 20-100, (iii) 40-50 and (iv) 100-20, where the first number represents epochs and the second is iterations per epoch. Our algorithm ends an epoch when a solution is found so the 2000 is an upper bound on the budget.

We use the number of patches found, the total runtime and a metric we call per patch time which is the total runtime divided by the number of patches found (it is undefined when no patches are found).

V. RESULTS

We now present the results for both of our research questions. Resulting patches and other experimental artifacts are available on our supplementary website (see https://github.com/LavaOps/CRNRepair/).

A. RQ1: CRNRepair Effectiveness

Table III shows data for the effectiveness of CRNRepair. The first column is the program mutant labeled S#no and H#no for subtraction (top) and hailstone (bottom). These match the IDs from Gerten et al. [5]. There is no program variant S2, because that one was determined to be an equivalent mutant. The next column shows the number of successful repairs (out of 10 epochs) followed by the percent successful. For instance, S1 finds a repair for all 10 epochs (100%) while S3 has 3 successful epochs with a (30% success). The next column shows the average number of iterations for successful repairs. This excludes failed repairs which all run to 100 iterations. The last column shows the total run time for the 10 epochs in minutes. This includes all epochs regardless of success

For subtraction, CRNRepair finds repairs for all faulty programs, except S5 (an overall success rate of 73%). We discuss S5 in more detail in Section VI. For 4 of the 9 programs: S1, S4, S8 and S9, we find a patch for all epochs. For all others we find a patch for at least half of the epochs. The average iteration where a solution is found ranges from 2.7 in S4 to 48.6 in S6. The runtimes shown in minutes for all 10 epochs range from 13 to 430 minutes.

For hailstone we find fewer patches. We find at least one patch for half of the subjects. H6 finds a patch for more than half of the epochs, while the others find a solution in less than half (1 or 3 epochs). The average iterations where a solution is found ranges from 9 to 40.7. Overall 14% of the epochs find a solution. Runtimes vary from 339 minutes to 650 minutes (or 10.8 hours).

We next look at results for the random search (shown in Table IV). For subtraction we see that random search has a similar success rate. The same set of four programs always finds a solution. For the others a few less were found (overall 63% of epochs found a patch). In hailstone random finds a solution only for H9 and for only 2 of the 10 epochs.

We calculated patch lengths, and see an average patch length of 1.4 and 3.1 respectively for subtraction for sim-

ulated annealing and random search. For hailstone these numbers are 1.1 and 1.5.

TABLE III
Count and percent of patches by Program Variant. This is our standard implementation using Simulated Annealing, 10 epochs of 100 iterations each.

ID	Num	Percent	Avg. Iters	Time
Subtraction				
S1	10	100	19.6	80
S3	5	50	47.2	324
S4	10	100	2.7	13
S5	0	0	–	430
S6	9	90	48.6	223
S7	7	70	44.6	254
S8	10	100	17.6	76
S9	10	100	10.1	37
S10	5	50	18.2	246
Hailstone				
H1	3	30	16.7	399
H2	3	30	39.0	427
H3	0	0	–	440
H4	6	60	40.7	335
H5	0	0	–	650
H6	0	0	–	408
H7	0	0	–	423
H8	1	10	9.0	345
H9	1	10	37.0	381
H10	0	0	–	517

TABLE IV
Count of Repairs by Program Variant using CRNRepair with random search

ID	Num	Percent	Avg. Iters	Time
Subtraction				
S1	10	100	21.9	83
S3	3	30	34.3	355
S4	10	100	12.5	52
S5	0	0	–	429
S6	6	60	50.3	321
S7	6	60	38.5	269
S8	10	100	17.7	69
S9	10	100	17.8	77
S10	2	20	39	392
Hailstone				
H1	0	0	–	596
H2	0	0	–	490
H3	0	0	–	311
H4	0	0	–	821
H5	0	0	–	849
H6	0	0	–	677
H7	0	0	–	420
H8	0	0	–	419
H9	2	20	52.5	294
H10	0	0	–	709

1) Patch Correctness: We test each patched program with its expanded test suite and confirm we find no failures. Once a patch passes, we manually inspect each to confirm it is plausible and characterized each by its repair type. Table V shows this data. We show only variants with at least one patch. The first column is the program variant. The next three columns are the repair types. The last column counts the number of patches that are validated as correct programs.

As we can see in subtraction, the patch types are split between finding a patch that creates the original program presented in Gerten et al. [12] (the golden patch). The other repairs are functionally correct, but have additional reactions that speed up the speed of computation. There are also 2 repairs that slowdown either the computation or the simulation. All patches are deemed correct.

Table VI shows similar data for the patches found by the random algorithm. We see a similar breakout of the types of repairs. All are determined to be correct.

TABLE V
Counts of types of repairs found by CRNRepair using simulated annealing. Golden repairs match the models from [5], Speedup increases the computation speed, Slowdown decreases the computation speed. No. Valid are the number of correct repairs.

ID	Golden	Speedup	SlowDown	No. Valid
Subtraction				
S1	7	3	0	10
S3	5	0	0	5
S4	3	7	0	10
S6	8	1	0	9
S7	5	2	0	7
S8	2	8	0	10
S9	2	7	1	10
S10	5	0	0	5
Total	37	28	1	66
Hailstone				
H1	3	0	0	3
H2	1	1	1	3
H4	5	1	0	6
H8	1	0	0	1
H9	1	0	0	1
Total	11	2	1	14

TABLE VI
Counts of types of repairs found by CRNRepair using random search. Golden repairs match the models from [5], Speedup increases the computation speed. No. Valid are the number of correct repairs.

ID	Golden	Speedup	No. Valid
Subtraction			
S1	7	3	10
S3	2	1	3
S4	5	5	10
S6	5	1	6
S7	4	2	6
S8	4	6	10
S9	6	4	10
S10	0	2	2
Total	33	24	57
Hailstone			
H9	1	1	2

Summary of RQ1. CRNRepair is able to find patches for all but one of the subtraction programs and half of the hailstone programs. Simulated annealing finds more patches than random search. All patches are determined to be plausible and correct.

B. RQ2: Epochs vs. Iterations

We now examine our experiments to understand the impact of epochs versus iterations. Table VII shows the four configurations. The next seven columns show the number of patches found by program variant. This is followed by the total time in minutes for all of the programs in that configuration to run. Next we show the total number of patches, followed by the percent of epochs that found a patch. The last column has the per-patch time. This is the total time divided by the number of patches found. We want to minimize this.

As we can see, except for H9, all configurations find some number of patches. The largest number of patches (136) is found by the 100 epoch, 20 iteration configuration. It also has the smallest per patch time (35.1 minutes), suggesting this is a good configuration. However, if we look at the percent of patches per epoch, it is only 19.4% suggesting that 20 iterations is likely insufficient. At the other extreme if we examine the 10 epoch, 200 iteration configuration, we have a high per-epoch success percent (64.3%), and the overall run time is the shortest, but the per-patch time is higher (69.0 minutes). This is also the only configuration that solved H9. We examined the logs, and found that these solutions were found at 44 and 150 iterations. Last we examine the 40 epoch, 50 iteration configuration. This seems to be a nice middle ground. We have a 38.2% per-epoch success rate, but the per-patch time is only 41.9 minutes. The per-epoch success is double that of the 100 epoch configuration and the per patch time is only 6 minutes more.

Summary of RQ2. We find that more epochs appear to reduce the per-patch time, while more iterations increase the chance of finding a patch per epoch. If we only consider per patch time, it is better to run more epochs than iterations, but if we want a high per epoch success rate, we need to increase the iterations. The 40 epoch, 50 iteration configuration is a good configuration for these programs.

VI. Discussion

First we examine the types of repairs made by CRN-Repair. If we examine Tables V and VI, we can see that more than 50% of the repairs result in the golden CRN as described in Gerten et al. [5]. We discuss some of the other types in more detail next.

Speedup manifests itself by adding two different types of reactions to the CRN. First, we see repaired CRNs with duplicate reactions,(e.g. S1_4). When this occurs, the probability of that reaction firing increases, and this decreases the overall simulation time.

The other mechanism we found that increases the overall speed of the CRN computation time involves reactions of the form $X1 + X2 \rightarrow null$. When a repaired CRN includes this type of reaction, such as in the S9 subject for subtraction, X1 and X2 are removed from the system in equal numbers. Since the CRN computes $X1 - X2$ and removing the same number of each does not change the result of subtraction, the CRN still correctly computes the difference. Depending on how large the initial molecule counts of $X1$ and $X2$, the computation time can be significantly faster.

We also saw some repairs that instead slow down the simulation and/or computation times. For example, a repair for the S9 subject added the reaction $X2 + Y \rightarrow X2 + X1$. This reaction moves a Y species back to a $X1$ species, effectively undoing the first reaction $X1 \rightarrow Y$ as long as an $X2$ species is present. Thus, a number of extra reactions must occur for the CRN to complete, lengthening the computation time. Note that the speed of reactions(determined by either rate constants or catalysts) can change not only the performance of the CRN, but also the correctness.

Repairs can also slow down the Matlab execution time as well as the computation time. It is arguable if reactions such as $X1 \rightarrow X1$ change the CRN, since it is easy to prove to have the same functional result as the CRN without it. However, repairs with this type add computation time (the time it takes to destroy $X1$ and then recreate it) and add Matlab execution time. The latter is caused by numerous CPU cycles required to simulate a reaction that does not move the computation towards completion, (e.g. S7_3).

While we were able to repair a large number of the programs studied, we also had some instances where we were unsuccessful. We now discuss some limitations and future directions based on observations about these instances.

Mutation Operators We found that some of the programs which did not repair (e.g. S5) are lacking in genetic material needed to create a fix with the existing mutation operators. The insert operator copies another reaction (in this case it only has reaction 2) and that reaction is of a different form from the one that is needed for repair. Simply moving and copying statements and replacing species is not enough to fix the program. Instead we need an operator to add (completely new) reactions. We plan an extension that will provide this functionality.

Fault localization. One of the important aspects of program repair is a good fault localization technique to guide the repair. In this work we lack this guidance. Given the nature of how a CRN program works, we expect most reactions to occur in all execution traces, hence, the traditional notion of coverage used in fault localization tools will not work. This is an interesting research direction. We plan to start by using reaction counting (number of times individual reactions occur) as a proxy for coverage.

Fitness. In this paper we have used a simple fitness, the count of failed simulations. However, we have observed large plateaus during search. We have considered more complex fitness functions that try to capture the spread across abstract tests and incorporate flakiness, but

TABLE VII
EPOCHS VS. ITERATIONS

Configs.		Patches Found by Program							Metrics			
Epochs	**Iter.**	**S3**	**S6**	**S10**	**H1**	**H2**	**H4**	**H9**	**Time (mins)**	**Tot. Patch**	**Percent**	**Patch Time(mins)**
10	200	4	10	8	6	8	7	2	3,104.6	45	64.3	69.0
20	100	6	15	13	6	10	14	0	3,698.1	64	45.7	57.8
40	50	15	23	16	11	25	17	0	4,487.1	107	38.2	41.9
100	20	20	41	13	10	27	25	0	4,775.3	136	19.4	35.1

without significant improvement. We believe this is an interesting future direction.

Test Flakiness. In our prior work, ChemTest [5], test flakiness (and mutant flakiness) was inherent in the testing process. We are not directly considering flakiness here. The correct CRNs are stable and should not exhibit this behavior, hence we are using the count of failed simulations to incorporate this metric (we run each simulation 10 times). But we believe there is interesting information that we are losing. We also want to support other types of CRNs that may not be stable. Allowing for flakiness during repair is an interesting direction that we believe has relevance for traditional repair frameworks as well.

AST for SBML. We use a standard SBML representation for this work (with minor additions as annotations). However, SBML is not hierarchical; it is flat. This means we lack much of the program structure that is contained in a traditional AST. We believe a refined SBML that provides a more modular view of the CRN, with more structural depenencies between reactions, may help improve localization and repair.

VII. Conclusions and Future Work

In this paper we have presented CRNRepair, a program repair framework for CRNs. We have built this on top of the PyGGI framework. We use the Matlab SimBiology library to calculate the fitness. We have evaluated this on 19 mutant programs from two CRNs and can repair 90% from the smaller CRN, and 50% from the other.

We have identified several interesting avenues for future work. First, we plan to build a reaction construction operator, that adds new reactions to the CRN. Second, we are adding additional search algorithms, and refining our existing ones. Third, we plan to explore the use of a hierarchical SBML which has a more refined AST structure. Fourth, we are working on alternative fitness functions. Last, we plan to run further experiments and evaluate on larger CRNs.

Acknowledgment

This work was funded in part by National Science Foundation Grants CCF-1909688, CCF-1901543 and FET-1900716

References

[1] D. Soloveichik, M. Cook, E. Winfree, and J. Bruck, "Computation with finite stochastic chemical reaction networks," *Natural Computing*, vol. 7, no. 4, pp. 615–633, 2008.

[2] S. J. Ellis, T. H. Klinge, and J. I. Lathrop, "Robust chemical circuits," *Biosystems*, p. 103983, 2019.

[3] R. Aris, "Prolegomena to the rational analysis of systems of chemical reactions," *Archive for Rational Mechanics and Analysis*, vol. 19, no. 2, pp. 81–99, 1965.

[4] D. Soloveichik, G. Seelig, and E. Winfree, "DNA as a universal substrate for chemical kinetics," in *International Meeting on DNA Computing*, ser. LNCS, vol. 5347, 2009, pp. 57–69.

[5] M. C. Gerten, J. I. Lathrop, M. B. Cohen, and T. H. Klinge, "ChemTest: An automated software testing framework for an emerging paradigm," in *International Conference on Automated Software Engineering (ASE)*, 2020, pp. 548–560.

[6] G. An, A. Blot, J. Petke, and S. Yoo, "PyGGI 2.0: Language independent genetic improvement framework," in *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE, 2019, pp. 1100–1104.

[7] M. Hucka, A. Finney, H. Sauro, H. Bolouri, J. Doyle, H. Kitano, A. Arkin, B. Bornstein, A. Cornish-Bowden, A. Cuellar, S. Dronov, E. Gilles, M. Ginkel, V. Gor, I. Goryanin, W. Hedley, C. Hodgman, J.-H. Hofmeyr, and J. Wang, "The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models," *Bioinformatics*, vol. 19, pp. 524–531, 03 2003.

[8] M. A. Gibson and J. Bruck, "Efficient exact stochastic simulation of chemical systems with many species and many channels," *The Journal of Physical Chemistry A*, vol. 104, no. 9, pp. 1876–1889, 2000.

[9] A. E. I. Brownlee, J. Petke, B. Alexander, E. T. Barr, M. Wagner, and D. R. White, "Gin: genetic improvement research made easy," in *The Genetic and Evolutionary Computation Conference*, A. Auger and T. Stützle, Eds., 2019, pp. 985–993.

[10] J. Petke, S. O. Haraldsson, M. Harman, W. B. Langdon, D. R. White, and J. R. Woodward, "Genetic improvement of software: A comprehensive survey," *IEEE Trans. Evol. Comput.*, vol. 22, no. 3, pp. 415–432, 2018.

[11] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Trans. Software Eng.*, vol. 38, no. 1, pp. 54–72, 2012.

[12] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Trans. Software Eng.*, vol. 45, no. 1, pp. 34–67, 2019.

[13] K. Wang, A. Sullivan, and S. Khurshid, "Automated model repair for alloy," in *International Conference on Automated Software Engineering*, ser. ASE, 2018, p. 577–588.

[14] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Commun. ACM*, vol. 62, no. 12, p. 56–65, Nov. 2019.

[15] M. L. Collard, M. J. Decker, and J. I. Maletic, "srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration," in *International Conference on Software Maintenance*, 2013, pp. 516–519.

[16] P. J. Van Laarhoven and E. H. Aarts, "Simulated annealing," in *Simulated annealing: Theory and applications*. Springer, 1987, pp. 7–15.

[17] MATLAB, *version 9.5.0 (R2018b)*. Natick, Massachusetts: The MathWorks Inc., 2019.