ChemTest: An Automated Software Testing Framework for an Emerging Paradigm

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

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

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

Titus H. Klinge
Drake University
Department of Mathematics and Computer Science
Des Moines, Iowa, USA
titus.klinge@drake.edu

ABSTRACT

In recent years the use of non-traditional computing mechanisms has grown rapidly. One paradigm uses chemical reaction networks (CRNs) to compute via chemical interactions. CRNs are used to prototype molecular devices at the nanoscale such as intelligent drug therapeutics. In practice, these programs are first written and simulated in environments such as MatLab and later compiled into physical molecules such as DNA strands. However, techniques for testing the correctness of CRNs are lacking. Current methods of validating CRNs include model checking and theorem proving, but these are limited in scalability. In this paper we present the first (to the best of our knowledge) testing framework for CRNs, ChemTest. ChemTest evaluates test oracles on individual simulation traces and supports functional, metamorphic, internal and hyper test cases. It also allows for flakiness and programs that are probabilistic. We performed a large case study demonstrating that ChemTest can find seeded faults and scales beyond model checking. Of our tests, 21% are inherently flaky, suggesting that systematic support for this paradigm is needed. On average, functional tests find 66.5% of the faults, while metamorphic tests find 80.4%, showing the benefit of using metamorphic relationships in our test framework. In addition, we show how the time at evaluation impacts fault detection.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

chemical reaction networks, software testing, metamorphic testing, flakiness

ACM Reference Format:
Michael C. Gerten, James I. Lathrop, Myra B. Cohen, and Titus H. Klinge. 2020. ChemTest: An Automated Software Testing Framework for an Emerging Paradigm. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3324884.3416638

1 INTRODUCTION

In recent years, utilization of non-traditional computing mechanisms (i.e. programs not written in declarative, imperative, functional languages) have proliferated in research and applications [2, 5, 8, 13, 14, 17, 29, 30, 47–51, 53, 56, 58–61, 63, 64]. Many of these paradigms can be specified with high-level programming languages [8, 18, 47, 52, 63]. One such paradigm is the chemical reaction network (CRN). CRNs are an abstraction of the traditional model of physical chemistry and are of special interest because they can be compiled into deoxyribonucleic acid (DNA) strands that simulate their behavior via strand displacement [4, 21, 61]. As a result, CRNs are used as a programming language to deploy *molecular programs* at the nanoscale. CRNs can naturally compute computational primitives such as addition, multiplication, and square roots [11, 15] as well as more complex algorithms such as watchdog timers, state logging, and finite automata [22, 23, 32]. With new technologies for synthesizing DNA and other molecules, it is now common to implement CRNs as physical nano devices in the lab. This powerful emerging computing paradigm is being promoted as a method to provide intelligent drug delivery and to achieve other computational functions at the nanoscale. Recently, new programming methods and tools have been developed to ease the development of molecular programs. There is even a new programming language, CRN++, designed to specify CRNs using traditional programming primitives and control structures [63].

Given the explosive growth in molecular systems, it is important to be able to validate and verify the behavior of CRNs. The potentially safety-critical nature of expected applications of this technology has led the research community to employ formal methods to prove correctness of stochastic CRNs via model checking, automated theorem proving, and even proving correctness by hand [21, 22, 34, 36, 39]. However, as noted in [39], model checking does not scale for large molecule populations. We demonstrate in our case study, that the PRISM model checker fails to build a model at a concentration of 50 molecules on at least one of our subjects.

Theorem proving techniques are scalable, but the dollar cost and time required to do so may be prohibitive, and the techniques are still rudimentary. While theorem proving techniques are justified for extreme safety-critical applications, there are increasing numbers of molecular applications that are less safety-critical, such as building DNA origami [51]. Therefore tolerating a small probability of failure is acceptable.

An alternative and common approach is to use software testing to improve scalability for validation, however, several issues make utilizing standard testing frameworks for testing CRNs challenging. First, to test CRNs we must rely on a simulation environment; testing the physical system involves extensive chemistry and can only occur for a limited set of instances (as in other cyberphysical environments). For instance, Qian et al. documented that a single experiment took 10 hours to run [48]. Second, as in many scientific systems, oracles and/or complete specifications may not exist. Third, the stochastic and asynchronous nature of these simulations means that they may result in flaky behavior obtaining different values for the same sets of inputs [7]. Fourth, even when a test is not flaky, it can be time dependent, and that may make it appear flaky even when it is not. Last, many of these programs can be probabilistic and sets of testing trials will be needed.

In this paper, we present the first (to the best of our knowledge) testing framework for CRNs, *ChemTest*. ChemTest supports different types of test cases (functional, metamorphic, internal and hyper). It formalizes requirements into an LTL-like language and then builds abstract (parameterized) test cases which can be instantiated for a range of input values. Oracle processing is performed at specific times on simulation traces of a CRN simulator. All tests are run multiple times to account for flakiness and multiple test trials are used for probabilistic programs.

In a large case study we evaluate CRNs of varying sizes and show that (1) we can specify constraints and oracles for test requirements, (2) all types of tests are effective in varying degrees and none outperform others on all mutants, (3) tests are dependent on time, and (4) test and mutant flakiness are inherent to the CRNs and must be accounted for. The contributions of this work are:

- An end to end approach for a new type of programming paradigm called ChemTest;
- A case study on different CRNs evaluating:
  - The need for different types of tests and
  - The importance of inherent test flakiness and probabilistic outcomes

In the next section, we present motivating examples and background on CRNs. In Section 3, we present ChemTest. We conduct a case study in sections 4 and 5 along with a discussion and roadmap for interesting research directions in CRN testing. We end with related work (Section 6) followed by conclusions and future work.

2 MOTIVATION AND BACKGROUND

Chemical reaction networks (CRNs) were first used to model and analyze chemical reactions over 50 years ago [4]. A CRN is composed of a set of species S and a set of reactions R over those species. A reaction is composed of a set of reactants (left side of reaction), a set of products (right side of reaction), and a rate constant that determines how fast the reaction proceeds. The law of mass-action yields one common semantic for CRNs and is divided into two variants: stochastic mass-action and deterministic mass-action semantics. In this paper we use stochastic mass-action semantics where molecule counts are natural numbers, and a Markov process determines the state of the network.

The probabilities are determined by the rate constant and product of the reactants. A simple example illustrates these concepts; most CRNs are much more complex than this. The CRN below consists of two reactions.

$$X1 \xrightarrow{1} Y \tag{1}$$

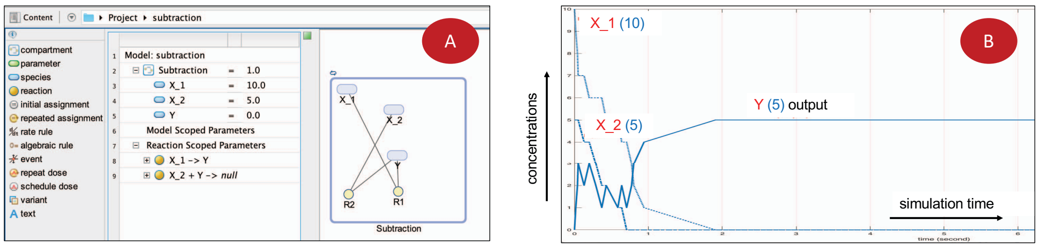
$$X2 + Y \xrightarrow{1} null \tag{2}$$

Reaction (1) converts species $X1$ to Y at a rate proportional to the product of the number of $X1$'s and the rate constant 1, and in reaction (2), 1 species Y is removed for every $X2$ in the system. This is a subtraction program ($X2$ is subtracted from $X1$ and placed in Y) that we use as one of our study subjects. This program cannot return negative values. If $X2$ is larger than $X1$ it outputs zero Y's.

We show a typical simulation environment for this CRN using MatLab's SimBiology package [42] in Figure 1. On the left (part A) we see the species at the top, initialized with molecular concentrations (we often just say molecules) of 10 and 5 for $X1$ and $X2$ respectively. Y starts with a concentration of 0. The reactions (R1 and R2) are on the bottom, and the model is shown on the right. On the right of this figure (part B) is a stochastic simulation of this CRN. At the start $X1$ is 10, while $X2$ is 5 and Y is 0. At 2 simulation seconds both $X1$ and $X2$ have reduced to 0, while Y is now 5. Y is the output variable for this CRN. We note that $X1$ and $X2$ are input variables, but we can also consider them internal variables since their value is not needed to evaluate the functional outcome. They are still important since in some cases a wrong internal state at the end could indicate a fault (we see this in our case study). This CRN is a specific type of a CRN which is called *stable*, since all of the reactions will stop once $X1$ and $X2$ have been fully utilized and the CRN will converge to a single specified state with probability 1. Thus the program is deterministic even though the CRN proceeds in a probabilistic fashion. Other CRNs may not stabilize to a single terminating state, and the program is expected to have a probabilistic result. We consider both types of programs in this work.

Verifying CRNs

The state of the art for verifying CRNs uses model checking to analyze specific properties in the state space given a (small) fixed number of initial molecules for each species [21, 22, 28, 34, 36, 39]. However, as seen in [22] on many real programs, this will not scale beyond 5 to 10 molecules. The subtraction CRN is relatively small, therefore we can model check it for 1000s of molecules. The goal of verification however, is to evaluate potentially buggy programs. If we add a single fault to this program, it is possible it will change our ability to scale model checking on this CRN. In our case study, several of our mutants caused a model checker to run out of memory before it could build the initial model at concentrations of 2,000 molecules. On another CRN, the original (correct CRN) was unable



```
Model: subtraction
  Subtraction   =   1.0
    X_1         =   10.0
    X_2         =   5.0
    Y           =   0.0
    Model Scoped Parameters
  Reaction Scoped Parameters
    X_1 -> Y
    X_2 + Y -> null
```

Figure 1: The subtraction CRN encoded in MatLab's SimBiology package. (A) shows the species $X1$, $X2$, and Y. This program will perform the subtraction 10-5=5. (B) shows a stochastic simulation where over time. At the start $X1$ is 10 and $X2$ is 5, while Y is zero. At 2 simulation seconds Y (the output) stabilizes at the final result of 5.

to scale beyond 50 molecules. Both instances allowed for 20G of memory. Instead, we can test the CRN, which should scale to larger concentrations, but to the best of our knowledge there is no existing systematic method to do this.

2.1 Challenges of Testing CRNs

To test a CRN, we first represent the CRN in a simulation environment such as MatLab's SimBiology package [42], Visual GEC [46], or Nuskell [5]. We also need to determine the input and output species, a set of test requirements and test cases. Once the simulation is run, all of the species still hold some value (all species are global), so we will need to check their ending values if integrated with other programs. If we know the functional output (which we do in this case) we can perform traditional functional testing.

Since CRNs often model physical processes, many programs have no known oracles. For instance, the *absence detector* part of the Molecular Watchdog Timer [22] is a CRN that checks for the lack of a signal (heartbeat) over a given time period. For this CRN we may need other types of tests such as Metamorphic tests [12, 40, 54]. These are often used to reason about relationships between two different runs of a program when the exact oracle is unknown. An example metamorphic test for subtraction is: if the first test input, $X1$, is greater than the second test input $X1'$, and $X2$ remains the same for both, then the output in Y' will be smaller than the output in Y. Although we do not specify what Y is, we know that the second output is smaller. Another need for metamorphic testing would be in the absence of formal specifications or if we have partial requirements. This may help us bypass the need of a complete set of functional tests. In this work we support both functional and metamorphic tests.

Inherent Flakiness

Since CRNs are distributed systems based on the laws of physics, we cannot control the order in which reactions fire, hence there is no direct analogy to a *thread* in traditional concurrent programs [62]. This can impact the ability of our tests to detect a fault and leads to *flaky tests*. We believe these are common in CRNs, therefore, we choose to accept that we can't fix them and mitigate the issue with multiple test runs. An example fault that is detected flakily changes subtraction reaction #2 (above) to $X1 + Y \longrightarrow null$. This has the impact of reducing $X1$ to Y and then using $X1$ to remove Y. This is incorrect behavior, but the output in Y is dependent on the order reactions fire. In a small pre-study we found this fault less than 50% of the time over ten test runs.

Another incorrect CRN for subtraction (a random fault seeded in our study, mutant #8), has an additional (third reaction), $null \longrightarrow X2 + X1 + Y$, that creates new molecules. Adding an additional reaction is a common mistake a CRN programmer may make. One of our tests which checks if subtraction works correctly when $X2 > X1$ (Y should be 0), will be correct intermittently. Suppose we start with $X2 = 200$, $X1 = 0$. Y also equals 0 at the start of the program. The first two reactions (original reactions) will not fire since these are already complete. However, the third reaction is free to fire at any point in time. When this fires the new program state is $X2 = 201$, $X1 = 1$, $Y = 1$. Reaction 2 now can fire removing an $X2$ and a Y ($X2 = 200$, $X1 = 1$, $Y = 0$). At this point, Y has the correct value (0) for the output and it would return a correct functional result. But as other reactions continue to fire, this will return to an incorrect state, cycling between the correct and incorrect value. This mutant was found to be flaky for 25% of the input values on this test case. These are mostly large input values for $X2$ (small values stabilized quickly), something that we cannot know ahead of time. We also must be cognizant of simulation time, since evaluation before the CRN stabilizes can also cause flakiness.

We address all of the issues raised in ChemTest. We allow for different types of tests (functional, metamorphic, internal) and use abstract tests instantiated with a broad range of concrete values. We consider evaluations at different simulation times, and run simulations (tests) multiple times. We also support *sets* of tests (or hyper tests) that are needed to evaluate probabilistic outcomes.

3 CHEMTEST

We now present ChemTest as shown in Figure 2. We begin with an existing CRN program and a set of requirements (or partial requirements). We then formulate the *test requirements* using LTL-like properties [6]. These properties are then used to create different types of *abstract test cases* which define the *input species* and the *oracle*. We use category partition (in the current framework we implement this with the test specification language TSL [44]) to generate concrete test cases. These are informed by constraints that come from the properties and the input species. We then perform testing using a stochastic simulation engine and *oracle processing*, and the final output is the result of testing. We describe each part of the process in more detail next.

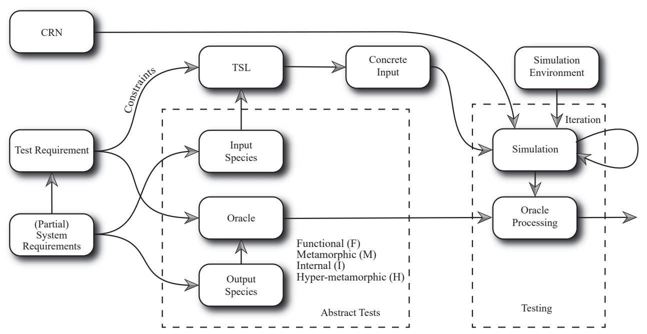


Figure 2: Overview of ChemTest. ChemTest starts with an existing CRN program and (partial) set of requirements. We formalize these and use the properties to create abstract test cases. We then generate concrete test cases, perform simulations and process the oracle.

3.1 Formalizing Test Requirements

In order to generate test cases and their oracles, we need a set of requirements to test. Since stochastic chemical reaction networks are modeled using Markov chains, temporal logic is a natural choice. *Linear temporal logic* (*LTL*) is a rich logic that classifies the paths of a Markov chain and is especially useful for our purposes.

The structure of an LTL formula ϕ can be defined recursively in the following Backus–Naur form:

$$\phi := \text{true} \mid a \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid X\ \phi \mid \phi_1\ U\ \phi_2. \tag{3}$$

Linear temporal logic formulas [6] specify constraints on infinite paths $\omega = (s_1, s_2, \ldots)$ through a Markov chain where each s_i is a *state*. In equation (3), a is an *atomic proposition* which evaluates to true if the first state s_1 of the path satisfies the proposition a; $X\ \phi$ says that ϕ is true in the *next* state of ω, i.e., that $(s_2, s_3, \ldots)$ satisfies ϕ; and $\phi_1\ U\ \phi_2$ says that ϕ_2 eventually holds starting at some future state s_i and that ϕ_1 holds for every state $s_1, \ldots, s_{i-1}$.

Two commonly used operators are *future* defined by $F\ \phi :=$ true $U\ \phi$ and *globally* defined by $G\ \phi := \neg F \neg \phi$. Intuitively, $F\ \phi$ is true if there exists a future state s_i that satisfies ϕ and $G\ \phi$ is true if every state s_i in the path satisfies ϕ.

In this work, we use an LTL-like notation, adding some new operators specific to CRNs. We leave the formalization and description of this notation for elsewhere, and instead describe important aspects of the notation as we go.

Consider the subtraction CRN defined earlier by the reactions in equations (1) and (2) which takes inputs $X1$ and $X2$ and produces a number of Ys equal to $X1 - X2$. Many of our test oracles for subtraction require that if the number of $X1$ input molecules change, then the number of Y output molecules change accordingly. One such test is as follows:

$$\left[X1'(0) = 2 \cdot X1(0) + 1\right] \wedge \left[X1(0) \text{ is even}\right] \rightarrow FG\left[Y' > Y\right]. \tag{4}$$

This is a metamorphic test which compares the behavior of the CRN on two different inputs. We denote one input with species $X1, X2$ and the other input with $X1', X2'$. This requirement specifies that when the number of initial $X1$ molecules, denoted by $X1(0)$, is increased by a specified amount, then the output Y' of the CRN with more $X1$ molecules will eventually be always greater than Y, the one with fewer $X1$ molecules.

We manually created these specification for this work. We provide additional information on this notation and all of our formalized specifications on our supplementary web page.

3.2 Abstract Test Generation

After we formalize test requirements, we generate abstract test cases. The input/output species are determined by the CRN. A test requirement specifies *constraints* (left side of implication) and the oracle (right side of implication). We define four types of abstract tests. The first are functional tests (F) which use a single set of inputs and have a known output. An example functional test for subtraction is

$$[X1(0) > X2(0)] \rightarrow FG\,[Y = X1 - X2], \tag{5}$$

which has a constraint that $X1$ is greater than $X2$ at the start of the program. The oracle states that Y will eventually always equal $X1 - X2$. The second type of abstract tests are metamorphic tests (M) which include two different input sets and are evaluated based on the relationship of the outputs. The requirement shown in equation (4) is an example of a metamorphic test. The third type of tests are internal tests (I), which check internal state of the CRN is correct at the end of computation.

$$[X2(0) > X1(0)] \rightarrow FG\,[X1 = 0], \tag{6}$$

This test in equation (6) tests the subtraction property when the initial number of molecules in $X2$ are greater than in $X1$. We expect the value of molecules in $X1$ (an internal variable) to be zero when the computation is complete. This is internal since it does not involve our output species (Y). It is a stronger program oracle and is important as we move towards integration testing, since other modules may depend on the state of this variable; The hailstone subject from our study demonstrates an example of an internal parity species which is a likely candidate for use by other modules.

The last type of tests are hyper-metamorphic tests which we refer to as simply *hyper* tests (H). A hyper test consists of multiple runs of the same metamorphic test used for probabilistic programs such as approximate majority, which is one of the CRNs we investigate in our case study. Hyper tests evaluate the result of metamorphic tests over some number of runs. An example of a hyper test is

$$\begin{aligned}&[X1(0) > X2'(0) > X2(0)] \\ &\quad \rightarrow \#[FG[X1' \text{ wins}]] < \#[FG[X1 \text{ wins}]].\end{aligned} \tag{7}$$

This test requirement states that, if the number of $X2$ molecules is increased, then $X1$ “wins” less frequently. Since approximate majority is an algorithm that determines which of two species has greater initial population by converting all molecules to a single species, we use “$X1$ wins” as shorthand to describe that species $X1$ has completely annihilated the population of $X2$ and therefore has the majority. The # operator counts the number of times the oracle is satisfied over many simulations of the CRN on that input.

3.3 Concrete Test Generation

We use constraints from the test requirements and the input species to generate concrete inputs using TSL. Each abstract test has one or more concrete test cases that partition its valid test space. As an example, in subtraction 10 and 5 can be concrete values for $X1$ and $X2$. We use partitions that include large, small, even, odd, etc. Our TSL is provided on the supplementary website.

3.4 Simulation

We simulate all of the CRNs using a stochastic simulator such as MatLab's SimBiology environment [42]. We run each test, N times (a parameter of our testing process). In our study we use 100 for N. We also select a simulation time (relative time used by the simulator). This may vary based on the CRN (see RQ2) and is important to allow the CRN to stabilize. We collect traces from the simulation to use in the oracle. For the hyper tests we run each of our N iterations T times. In our study we use 10 for T.

3.5 Oracle Processing

Last, we evaluate the results of checking each test requirement against the system traces. We implemented a library to check properties against the simulation traces. We first read the full trace and examine the program state for *each* time interval. This is computationally expensive, but complete. For the metamorphic tests we evaluate two traces (the two different input simulations) together and for hyper tests we run the analysis on pairs of simulations, T times and count the number of times the requirement holds. We note that our *future globally* operator may fail past the time that we are evaluating, but we assume the correctness within the evaluation time.

4 CASE STUDY

We evaluate several facets of ChemTest. Supplemental data for our experiments are found on our supplementary website.[1] We ask the following three research questions in this study. The first question focuses on ChemTest's core effectiveness:

RQ1. How effective is ChemTest at fault detection?

As part of this question we ask how well ChemTest scales by comparing it against the state of the art, model-checking. The next two questions focus on unique aspects of CRNs

RQ2. What is the impact of time on simulations?

RQ3. How do the stochastic and probabilistic aspects of CRNs impact test results?

4.1 Objects of Study

We have selected three commonly used CRNs to study in this paper. The first two are often used to illustrate CRN behavior. The third subject has been used in many research papers [16, 31]. The first, subtraction, is a simple CRN (only two reactions and 3 species) that has an obvious functional output. The second, hailstone, is more complicated with 11 reactions and 11 species. The third is a common CRN, approximate majority, described earlier, that has a probabilistic output. It returns the correct result a large percentage of time, but is not guaranteed to always converge on the same answer. This has only 4 reactions and 3 species. Table 1 shows the reactions for each of these programs. We describe each in more detail below.

[1] https://github.com/LavaOps/ChemTest

Subtraction. This CRN computes $f(n_1, n_2) = n_1 - n_2$ using input species X_1, X_2 and output species Y. However, since CRNs cannot have *negative* molecule counts, the CRN outputs 0 if $n_1 < n_2$.

Hailstone. This CRN computes the hailstone function [35]

$$f(n) = \begin{cases} n/2, & \text{if } n \text{ is even} \\ 3n+1, & \text{if } n \text{ is odd} \end{cases}$$

using input species $X1$ and output species Y.

Approximate Majority. This CRN models a probabilistic algorithm that is used in nature to make binary decisions such as the cell cycle switch. It will quickly decide which of two species has more molecules. Given an initial population of $X1$ and $X2$ the algorithm outputs its decision by converting the total population of molecules to the species with the initial majority. Both $X1$ and $X2$ are outputs.

Table 1: Reactions defining subject programs

Subtraction (S)	Hailstone (H)	Approximate Majority (AM)
$X1 \rightarrow Y$	$X1 \rightarrow PO + H + M$	$X1 + X2 \rightarrow U + X1$
$X2 + Y \rightarrow null$	$PO + PO \rightarrow PE$	$X1 + U \rightarrow X1 + X1$
	$PE + PO \rightarrow PO$	$X2 + U \rightarrow X2 + X2$
	$PE + PE \rightarrow PE$	$X1 + X2 \rightarrow U + X2$
	$H + H \rightarrow D$	
	$M \rightarrow 3B + 6A$	
	$2B + 2A \rightarrow null$	
	$PE + D \rightarrow PE + CE + Y$	
	$PO + A \rightarrow PO + CO + Y$	
	$CE + PO + Y \rightarrow PO + D$	
	$CO + PE + Y \rightarrow PE + A$	

4.2 Fault Seeding

Since we don't have an existing bug repository of faulty CRNs, we generate random program mutants (i.e. we use mutation testing). Program mutants have been shown to be similar to common types of faults in traditional programs [3, 43]. While we cannot guarantee these are realistic and/or sufficient, we generate mutants that are first order mutants (i.e. single changes) and have restricted them to similar types of faults we have seen in the CRN programs we have studied. The mutants are generated as follows. For each mutant we randomly select a reaction and randomly select from a set of high level operations, (1) add a new reaction, (2) remove a reaction, (3) change a reaction. To add a new reaction we select from one of 15 templates (up to three reactants and three products) and then for each of the species we assign a valid species from the program (at random). To change a reaction we choose to either add a product/reactant, remove a product/reactant or change an existing product/reactant to a different species from that CRN.

We generated 10 random mutants for each subject. For approximate majority, one of the mutant simulations timed out after 4 days of runtime, therefore it was removed. We also restricted the types of reaction modifications for approximate majority to preserve the existence of two reactants in all cases, because this changes the rate (a parameter of simulation) and we discovered that SimBiology does not correctly handle different rates (its default is 1). This was not an issue for other subjects since they are stable CRNs. Table 2 shows the mutants for each subject by number. The changed/added reaction is shown followed by a reaction number. If the number is larger than the number of reactions in the original CRN (e.g. subtraction, M1) this means a new reaction is added. All others represent changes and/or removals.

4.3 ChemTest Implementation

We manually created program requirements which formed our abstract test cases. These can be found on our website. We generate concrete test cases for all abstract test cases using TSL to define partitions for each CRN (such as even, odd, large, small), guided by test requirement constraints. Since some of our abstract tests have constraints such as $X1 > X2$, each set of abstract tests has a different number of concrete tests. For all CRNs we used 200 as the maximum input value for a single species. In some of our metamorphic tests, the second trace required a species that is larger (by a factor), e.g. test number 6 for hailstone, hence our largest input population can be as high as 401 molecules.

We run all concrete tests 100 times. If a concrete test has a single trace (i.e. functional, internal) we have 100 traces for the test case. In the case of metamorphic tests there are two traces for each concrete test, hence we have 200 traces. For the hyper-metamorphic tests we repeat our tests 10 times, therefore we have 1000 test runs and 2000 traces (2 traces for each test). We run all tests (non interactively) on a heterogeneous computation cluster with an allocation of 20GB of RAM, Intel CPU with frequencies from 2.1 - 3.5 GHz and utilizing a single processor core, running Red Hat Linux 7 and Mabtlab2019a version R2019a-io4754x. All rate constants are kept at 1 for all reactions in the subjects of this study.

After simulation, we run the evaluation script on the generated simulation traces. The oracle processing is done using a Python script. It evaluates the properties on the CRN simulation, returning (for each trace iteration) whether or not the property holds.

4.4 Flakiness Metrics

We define the metrics used to differentiate deterministic and flaky tests and mutants in RQ3.

Deterministic/Flaky Concrete Test: A concrete test is *deterministic* if it fails on all N simulation traces during oracle processing and is *flaky* if it fails on at least one but not all N traces.
Deterministic/Flaky/Mixed Abstract Test: An abstract test is *deterministic* if all of its concrete tests are deterministic, is *flaky* if all of its corresponding concrete tests are flaky, and is *mixed* it has a combination of deterministic and flaky concrete tests.
Deterministic/Flaky/Mixed Mutant: A mutant is *deterministic* if every test that finds it is deterministic, is *flaky* if every test that it finds it is flaky, and *mixed* if there are both deterministic and flaky failures.

4.5 Threats to Validity

We outline the most important threats to validity here. With respect to external validity (generalization) we only used three CRNs. However, we used CRNs that have different characteristics, used for different purposes. We also ran all of our simulations using MatLab's SimBiology package. We did, however, keep oracle processing as a separate program so that this can be used on alternative types of simulation traces. While we believe that ChemTest will work for other stochastic simulation engines, we leave this as future work. With respect to internal validity, the authors of this paper wrote the requirements definitions. We tried to use common properties of the systems we were testing, but, we cannot be sure that they are complete and/or representative of what other's might develop. We leave automated test generation (from the CRN model itself) as future work. All of our analysis used automated programs which could have faults themselves. We selected subsets of our data to validate by hand, and examined multiple individual faults in depth. We have also provide artifacts for this work on an external website for others to re-validate. With respect to construct validity, the use of correct metrics, we acknowledge there may be better metrics to use, but we chose standard metrics (such as fault detection and runtime) used in testing.

5 RESULTS

In this section we present the results of each of our research questions. We follow with a discussion of some interesting observations, and end with a roadmap for the future of CRN testing.

5.1 RQ1: How effective is ChemTest at fault detection?

Table 3 shows testing results for the subtraction (top) and hailstone subjects (bottom). Table 4 shows data for the approximate majority subject. The first column is the abstract test ID, the second is the number of concrete tests generated for that test. The next column states the test type where "F" means functional, "M" means metamorphic, "I" means internal, and "H" means hyper test. The columns represent individual mutations (10 for subtraction and hailstone, and 9 for approximate majority). Each test ID has two rows. The first (unshaded) is the percent of failing tests that fail on all 100 runs (i.e. deterministic). The second (shaded row) indicates the percent of failed tests that fail in at least one, but not all of the 100 runs, (i.e. flaky). The sum of the two rows indicates the percent of tests failing for that abstract test. For example, in the first row of subtraction, we see that this is a functional abstract test with 40 concrete tests. For mutation 1, 87% of the abstract tests fail deterministically and none are flaky. On the other hand, for mutation 3, 10.9% are flaky, 87% are deterministic; the total detection rate is 97.9%. Both of these mutants are easily detectable with this functional test, however, for mutation 3 the concrete test plays a bigger role. It is possible to miss the fault depending on the input and number of runs.

Overall, we can see that all mutants, except subtraction mutation 2, are detected. We examined mutation 2 and determined it is an equivalent mutation, hence we remove this from the rest of the analyses (i.e. RQ2 and RQ3). However, mutation 2 did cause performance problems during our simulations since it is creating additional (unneeded reactions) accounting for 78% of the simulation time at time 100. For subtraction, all mutants are found by a mixture of functional, metamorphic, and internal tests and are found both deterministically and flakily.

For hailstone, mutation 1 is only found by a single functional test, however, all three concrete tests detect the fault deterministically.

Table 2: Mutants by subject. For each subject the changed reaction (R) and reaction number (#) is given.

S	Change	R #	H	Change	R #	AM	Change	R #
$S1$	$X1 + X1- > X2$	3	$H1$	$2B + 3A- > null$	7	$A1$	$X1 + X2- > U + X2 + U$	4
$S2$	$Y- > Y$	3	$H2$	$M- > H$	12	$A2$	$Removed$	1
$S3$	$X1 + Y- > null$	2	$H3$	$CE- > 3B + 6A$	6	$A3$	$X1 + X2- > U + U$	1
$S4$	$X1 + X2 + Y- > null$	3	$H4$	$CE + PO + y- > null$	12	$A4$	$Removed$	2
$S5$	$Removed$	1	$H5$	N/A	4	$A5$	$X2 + X1- > X2 + X2$	3
$S6$	$X2 + Y + X1- > null$	2	$H6$	$X1- > PO + H$	1	$A6$	$X1 + X2- > U + U + X2$	4
$S7$	$Y- > null$	2	$H7$	$CO + PE + Y- > A$	11	$A7$	$X1 + X2- > U + X1 + U$	1
$S8$	$null- > X2 + X1 + Y$	3	$H8$	$PO + A + Y- > PO + CO + Y$	9	$A8$	$X1 + U- > null$	2
$S9$	$Y- > X2 + X2$	3	$H9$	$2B + 2A + PO- > null$	7	$A9$	$U + U- > X1 + X1$	2
$S10$	$X2- > null$	2	$H10$	$CO + PE + Y + CE- > PE + A$	11			

Mutation 5 is only found by an internal test. Mutation 6 is only found by one of the functional tests, but multiple metamorphic tests (this pattern is reversed in other mutants). One of the functional tests does not find any of the mutants, and no test finds all mutants.

Next we turn to the approximate majority (Table 4). Since this is probabilistic, we expect some failures in the original program. We include an additional column (column 3), labeled "O," which is the original, non-mutated CRN. In all tests, we only see flaky failures. For most of the mutations, the mutants fail at a higher rate than the original in at least some tests.

We now look at the runtimes for the experiments. We capture the runtime for all 100 simulations of each concrete test and the time taken to evaluate the oracle. This data is presented in Table 5. (We include the data for subtraction mutation 2 in this calculation.) For subtraction, the simulation time took 5.2 hours and the oracle analysis took 24.6 hours. For hailstone, this is 5.3 hours and 16.8 hours respectively. For approximate majority, the times rose to 15.0 days and 42.2 days. Overall, the MatLab simulation time accounts for 17-26% of the total testing time, thus the majority of the testing time was due to oracle processing on the simulation data. Part of the reason for the long oracle processing was due to the I/O needed to process the large, uncompressed, simulation files. Another reason is that our oracle processing library was written in Python and one of the LTL-like operators used an inefficient Python loop. Performance profiling revealed that the bottleneck was this loop and can be optimized by rewriting the library in a language like C. As future work we plan to optimize the analysis part of this study and build the oracle processing directly in MatLab. All the data collected in this study, which involves additional processing of the oracles for different time slots in RQ2 and RQ3, have used from 6.5 days to 310.4 days. Overall, the experiments run represent approximately a year of machine time, 25.5 days of which is simulation time.

Scalability. We compare against the current state of the art in validating CRNs, model checking. We selected the Probabilistic Symbolic (PR) Model Checker[33], Version 4.6. PR has previously been used to evaluate CRNs [34]. Since the first step of model checking, building the model, is required to evaluate individual properties, we focus on this step. Likewise, we focus on the simulation (and trace collection) phase of ChemTest for all concrete tests. It should be noted that both model checking and ChemTest can evaluate multiple properties on the same model or trace respectively. We used the default configuration of PR except for the cudd memory. We set this parameter `-cuddmaxmem` to 20g to give it a fair chance. This

Table 3: Percent of failing test cases by mutant. ID is the abstract test number, NT is the number of concrete tests, TT is the type of test, F-functional, M-metamorphic, I-internal. Columns represent mutations.

ID	NT	TT	1	2	3	4	5	6	7	8	9	10
Subtraction												
1	46	F	87.0	0.0	87.0	47.8	87.0	50.0	87.0	100.0	87.0	58.7
			0.0	0.0	10.9	0.0	0.0	8.7	0.0	0.0	0.0	0.0
2	40	F	0.0	0.0	25.0	0.0	0.0	25.0	0.0	70.0	0.0	55.0
			0.0	0.0	30.0	0.0	0.0	25.0	0.0	27.5	0.0	0.0
3	40	M	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
			0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	40	M	75.0	0.0	0.0	35.0	95.0	5.0	85.0	0.0	92.5	0.0
			17.5	0.0	97.5	10.0	0.0	45.0	15.0	75.0	5.0	2.5
5	40	M	85.0	0.0	0.0	42.5	90.0	5.0	90.0	0.0	90.0	90.0
			10.0	0.0	92.5	2.5	0.0	35.0	0.0	70.0	0.0	5.0
6	40	M	55.0	0.0	0.0	10.0	55.0	2.5	52.5	0.0	55.0	47.5
			45.0	0.0	52.5	5.0	0.0	17.5	0.0	57.5	0.0	5.0
7	46	M	0.0	0.0	6.5	0.0	0.0	10.9	0.0	76.1	0.0	58.7
			0.0	0.0	52.2	0.0	0.0	21.7	0.0	23.9	0.0	0.0
8	40	I	0.0	0.0	0.0	0.0	55.0	0.0	0.0	0.0	0.0	0.0
			0.0	0.0	0.0	0.0	0.0	0.0	0.0	100.0	0.0	0.0
9	40	I	55.0	0.0	55.0	55.0	55.0	55.0	55.0	100.0	0.0	100.0
			0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	55.0	0.0
Hailstone												
1	7	F	0.0	71.4	0.0	71.4	0.0	0.0	85.7	85.7	0.0	85.7
			0.0	14.3	0.0	14.3	0.0	0.0	0.0	0.0	85.7	0.0
2	3	F	100.0	100.0	100.0	0.0	0.0	100.0	0.0	100.0	100.0	100.0
			0.0	0.0	0.0	100.0	0.0	0.0	0.0	0.0	0.0	0.0
3	1	F	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
			0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	6	M	0.0	0.0	0.0	0.0	0.0	0.0	0.0	100.0	0.0	0.0
			0.0	0.0	0.0	100.0	0.0	0.0	0.0	0.0	100.0	0.0
5	7	M	0.0	0.0	14.3	0.0	0.0	100.0	0.0	100.0	14.3	0.0
			0.0	14.3	42.9	85.7	0.0	0.0	0.0	0.0	85.7	0.0
6	3	M	0.0	0.0	0.0	0.0	0.0	100.0	0.0	100.0	0.0	0.0
			0.0	0.0	100.0	100.0	0.0	0.0	0.0	0.0	66.7	0.0
7	3	M	0.0	0.0	0.0	0.0	0.0	100.0	100.0	100.0	0.0	100.0
			0.0	100.0	100.0	100.0	0.0	0.0	0.0	0.0	100.0	0.0
8	9	I	0.0	0.0	0.0	0.0	0.0	0.0	66.7	0.0	0.0	0.0
			0.0	0.0	0.0	100.0	66.7	0.0	0.0	0.0	100.0	0.0

Table 4: Percent of Mutants Found by Test Case for Approx. Majority. ID is the abstract test number, NT is the no. of concrete tests, TT is test type, F-functional, I-internal, H-hyper. Columns are mutations.

ID	NT	TT	O	1	2	3	4	5	6	7	8	9
1	190	F	0.00	0.72	0.00	0.00	0.04	0.34	0.72	0.71	0.59	0.00
			0.00	0.06	0.00	0.01	0.17	0.43	0.05	0.07	0.18	0.63
2	190	F	0.00	0.66	0.74	0.18	0.74	0.09	0.68	0.76	0.75	0.00
			0.34	0.12	0.00	0.34	0.00	0.68	0.09	0.02	0.03	0.77
3	183	F	0.00	0.73	0.00	0.00	0.00	0.68	0.73	0.64	0.39	0.01
			0.28	0.00	0.00	0.01	0.03	0.09	0.00	0.09	0.36	0.52
4	183	I	0.00	0.00	0.00	0.00	0.04	0.35	0.00	0.00	0.00	0.00
			0.00	0.00	0.00	0.01	0.16	0.42	0.00	0.00	0.09	0.62
5	183	H	0.00	0.00	0.00	0.10	0.01	0.30	0.00	0.00	0.00	0.00
			0.04	0.36	0.12	0.08	0.23	0.40	0.05	0.26	0.00	0.75
6	183	H	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
			0.02	0.17	0.00	0.01	0.00	0.18	0.10	0.00	0.00	0.77
7	183	H	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
			0.02	0.00	0.00	0.01	0.03	0.00	0.00	0.00	0.00	0.02
8	183	H	0.00	0.00	0.00	0.13	0.00	0.19	0.00	0.01	0.00	0.00
			0.11	0.34	0.38	0.30	0.49	0.54	0.07	0.27	0.01	0.73
9	183	H	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
			0.09	0.16	0.00	0.07	0.00	0.40	0.10	0.00	0.00	0.74
10	183	H	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
			0.09	0.00	0.00	0.07	0.10	0.00	0.00	0.01	0.01	0.12
11	183	H	0.00	0.27	0.68	0.46	0.46	0.40	0.09	0.24	0.00	0.00
			0.22	0.18	0.09	0.11	0.09	0.34	0.03	0.13	0.03	0.76
12	183	H	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01
			0.08	0.02	0.00	0.04	0.00	0.21	0.00	0.00	0.00	0.75
13	183	H	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
			0.08	0.00	0.00	0.04	0.02	0.01	0.00	0.01	0.03	0.12
14	183	H	0.00	0.37	0.69	0.21	0.21	0.31	0.21	0.32	0.00	0.00
			0.14	0.23	0.02	0.08	0.07	0.32	0.07	0.17	0.01	0.67
15	183	H	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.07
			0.05	0.01	0.00	0.02	0.00	0.16	0.01	0.00	0.00	0.65
16	183	H	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
			0.05	0.00	0.00	0.02	0.03	0.01	0.00	0.01	0.01	0.04
17	183	H	0.00	0.01	0.00	0.00	0.00	0.00	0.01	0.00	0.00	0.00
			0.36	0.15	0.00	0.40	0.00	0.76	0.09	0.00	0.00	0.77
18	183	H	0.00	0.00	0.00	0.00	0.02	0.00	0.00	0.00	0.00	0.00
			0.36	0.00	0.00	0.40	0.38	0.00	0.00	0.01	0.02	0.17
19	183	H	0.00	0.01	0.01	0.01	0.01	0.01	0.01	0.02	0.01	0.01
			0.37	0.17	0.00	0.40	0.00	0.75	0.10	0.00	0.00	0.75
20	183	H	0.00	0.00	0.00	0.00	0.01	0.00	0.00	0.00	0.00	0.00
			0.37	0.00	0.00	0.40	0.41	0.01	0.00	0.01	0.02	0.20
21	183	H	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
			0.34	0.01	0.00	0.01	0.00	0.16	0.00	0.00	0.00	0.55
22	183	H	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.01
			0.34	0.00	0.00	0.01	0.02	0.12	0.00	0.17	0.42	0.56
23	183	H	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
			0.37	0.01	0.00	0.01	0.00	0.19	0.00	0.00	0.00	0.56
24	183	H	0.00	0.01	0.00	0.00	0.00	0.01	0.01	0.01	0.01	0.00
			0.37	0.00	0.00	0.01	0.01	0.09	0.00	0.17	0.43	0.58

is comparable with the 20G of memory we used to run ChemTest experiments. We gave PR 21 GB of RAM to account for overhead and use the same computing cluster as ChemTest. We use two of our CRNs, subtraction and Hailstone. For subtraction we started with an input size 40 and scaled up to 10k. For Hailstone we construct the model starting at 10 molecules, increasing by 10 to 100, where model checking regularly fails. We run for all mutants for each programs and record the time taken to build the models (or run the ChemTest simulations). We used a 6 hour timeout for both PR and ChemTest.

Table 5: RunTime Data. Total runtime for 100 simulations (Sim) for all concrete tests, time to calculate the oracle (Oracle). Times in Hours (h) or Days(h)

Subjects	Sim	Oracle	Total	% Sim	Tot. Exp. Times
S	5.2h	24.6h	29.8h	17.4	9.4d
H	5.3h	16.8h	22.1h	24.0	6.5d
AM	15.0d	42.2d	57.2d	26.2	310.4d

Table 6 shows a subset of the results (the rest is on our website). The rows represent input sizes and the columns are the mutant programs. The first column, O, is the original CRN. For each, we show the results in seconds, minutes(m) or hours (h) for each PR (PR) and ChemTest (CT). As we see in subtraction, PR is faster at small input sizes, but is not able to scale to 10k molecules in 8 of the 11 mutants. In Hailstone, PR fails to build a model for 10 of 11 models with an input of 100 molecules; demonstrating a loss of scalability on more complex CRNs. We explored Hailstone mutation 6 further. The largest input it handled has 220 molecules and consists of 1.7 billion states and 10 billion transitions.

Summary of RQ1. All four test types (metamorphic, internal and hyper tests) are effective at finding faults. Every mutation was identified by at least one test type, and the majority were found by multiple test types. We also see a mixture of deterministic and flaky detection across the various types of tests. With respect to scalability, we see that we can collect test traces in minutes, while PR fails to build models for larger molecule counts.

5.2 RQ2. What is the impact of simulation time on test results in ChemTest?

For this RQ we look at failures at simulation intervals for subtraction and hailstone. Approximate majority is on our website and shows similar results. We evaluate the oracle at time 2, 4, 6, 8, 10, 25, 50, 75, and 100 (the time used in RQ1). These times are internal Matlab simulation times and not relative simulation runtimes, i.e. the time chosen may have little impact on the practical runtimes, however, it can impact the length of the traces for analysis if we do not allow the simulation to run long enough.

Figure 3 shows the time data as box plots for two of our subjects. For each time interval (x-axis) we plot the number of failing concrete test cases per mutation. A failure means the test failed at least once for a mutation (this considers both flaky and deterministic failures). The red line is the original (correct) CRN. In all cases at time 2, the original CRN is appearing faulty since it has not yet converged on an answer. Over time this converges to zero failures. The box plots show that the number of failing tests drop over time and stabilize. We break out the deterministic and faulty failures in RQ3.

Summary of RQ2. We conclude that simulation time is very important. All three subjects are unstable early on, but converge at some point in time.

Table 6: Left is PRISM model checker (PR), right is time to run all tests in ChemTest (CT). Timeout of 6 hrs with 21 GB of memory. Times in seconds unless noted, minutes (m), hours (h). M is a memory error and T is a time out.

	Subtraction																					
Input	S0		S1		S2		S3		S4		S5		S6		S7		S8		S9		S10	
	PR	CT	PR	CT	PR	CT	PR	CT	PR	CT	PR	CT	PR	CT	PR	CT	PR	CT	PR	CT	PR	CT
40	0.1	1.9	0.1	8.5	0.1	2.1	0.1	9.5	1.0	15.8	0.0	8.7	1.1	17.2	0.0	10.0	12.7	13.0	0.1	9.6	0.0	8.3
200	0.2	3.0	0.4	10.4	0.3	3.1	0.1	11.4	2.3	9.4	0.0	8.9	1.0	9.7	0.1	11.2	11.5	14.8	0.4	11.1	0.2	9.7
400	1.5	4.6	2.4	10.1	1.6	4.7	0.5	13.5	19.19	11.8	0.1	7.8	13.5	10.7	0.3	13.8	1.8 m	16.9	2.8	13.8	1.3	10.8
600	4.6	6.3	6.4	11.5	7.3	6.2	1.3	16.4	1.32 m	14.6	0.4	7.5	1.1 m	13.6	0.7	17.6	7.3 m	23.2	5.2	17.4	2.7	14.9
800	8.9	7.9	12.6	14.1	9.6	7.8	2.8	17.7	2.76 m	15.2	0.4	8.5	2.2 m	15.7	1.3	23.3	14.1 m	23.1	9.9	20.1	5.0	16.6
1k	14.1	9.6	20.1	11.4	15.1	9.5	4.9	19.4	5.9m	16.5	0.8	7.6	4.9m	15.9	2.2	23.3	27.7m	26.1	16.2	24.5	8.0	19.7
2k	1.3m	24.2	1.6m	19.4	1.3m	17.4	24.9	29.5	M	23.5	4.0	9.2	M	22.8	10.3	36.5	2.6h	38.4	1.6m	36.3	51.3	35.3
3k	3.3 m	25.6	4.7 m	15.6	3.6 m	28.1	59.7	37.8	M	31.8	9.2	8.4	M	29.9	25.0	52.5	T	48.9	3.8 m	48.4	2.0 m	45.5
4k	5.9m	33.7	8.0m	19.0	6.3m	2.6m	1.8m	50.4	M	36.5	18.2	7.4	M	36.6	42.2	55.1	T	58.1	6.9m	57.0	3.5m	1.0m
5k	11.2 m	41.9	17.1 m	22.3	11.8 m	41.2	3.1 m	56.5	M	42.3	28.9	8.3	M	41.7	1.5 m	1.2 m	T	1.2 m	24.8 m	1.2 m	9.2 m	1.1 m
6k	16.3m	50.5	25.5m	29.1	17.0m	49.7	4.6m	1.1m	M	51.8	38.4	8.5	M	48.1	2.0m	1.4m	T	1.3m	22.4m	1.4m	10.2m	1.4m
7k	25.9 m	58.0	42.4 m	2.5 m	28.0 m	57.1	6.0 m	1.3 m	M	55.7	57.7	7.5	M	57.1	2.7 m	1.4 m	T	1.5 m	M	1.4 m	13.5 m	1.4 m
8k	29.3m	1.2m	1.8h	42.4	39.3m	5.2m	11.1m	1.3m	M	1.2m	1.5m	7.9	M	1.1m	3.7m	1.8m	T	1.9m	M	1.8m	17.5m	1.7m
9k	M	1.4 m	M	47.8	M	1.2 m	25.2 m	1.4 m	M	60.0	1.4 m	9.3	M	58.8	5.6 m	1.9 m	T	1.9 m	M	1.9 m	23.6 m	1.9 m
10k	M	1.4m	M	2.7m	M	2.5m	M	1.6m	M	1.2m	2.2m	13.2	M	1.2m	6.4m	2.1m	T	2.0m	M	2.0m	31.2m	2.0m
	Hailstone																					
Input	H0		H1		H2		H3		H4		H5		H6		H7		H8		H9		H10	
	PR	CT	PR	CT	PR	CT	PR	CT	PR	CT	PR	CT	PR	CT	PR	CT	PR	CT	PR	CT	PR	CT
10	2.6m	27.6	2.6	24.1	2.6m	22.7	21.4	8.4	46.7	11.1	36.3	15.2	19.7	6.1	29.7	8.5	38.6	10.8	1.4m	9.3	2.8m	2.2m
20	5.3m	50.9	5.0m	40.0	9.1m	41.8	32.6	2.2m	8.4m	19.6	3.4m	4.5m	12.3	9.0	5.4m	12.6	5.7m	16.8	16.1m	9.0	6.8m	13.7
30	26.8 m	1.4 m	24.3 m	59.4	M	1.1 m	4.4 m	19.6	3.2 m	30.0	44.0 m	1.0 m	15.7	13.5	44.4 m	27.2	56.6 m	23.9	3.3 h	12.13	45.8 m	2.3 m
40	2.5h	3.9m	2.2h	3.1m	M	1.6m	7.9m	26.6	M	48.8	2.9h	1.5m	30.7	18.7	3.2h	2.4m	3.9h	33.6	M	15.6	3.3h	24.9
50	M	2.6 m	M	1.8 m	M	2.1 m	55.1 m	34.8	2.9 m	2.9 m	M	4.2 m	42.0	33.9	3.5 h	2.5 m	M	38.1	M	19.0	T	30.5
60	M	3.3m	M	2.2m	T	2.6m	M	42.5	M	5.2m	M	4.8m	38.4	4.5m	T	32.8	T	45.7	M	29.2	T	38.7
70	M	4 m	M	2.7 m	T	5.0 m	M	51.2	M	1.4 m	M	9.4 m	1.1 m	36.7	T	40.2	T	3.0 m	M	26.3	T	41.8
80	M	6.8m	M	5.2m	M	3.8m	M	3.0m	M	1.6m	M	4.4m	1.6m	47.1	M	42.7	M	3.1m	M	2.5	T	53.2
90	T	7.4 m	T	5.7 m	T	6.5 m	M	1.2 m	M	5.9 m	M	8.9 m	2.4 m	57.5	M	53.2	T	3.1 m	M	33.6	T	52.9
100	M	6.2m	T	4.4m	T	5.1m	M	3.3m	M	2.2m	M	7.7m	3.5m	59.2	M	52.8	T	1.2m	M	45.7	T	3.1m

5.3 RQ3. How do the stochastic and probabilistic aspects of CRNs impact test results?

If we return to Table 3 and 4, we see a mix of tests failing either deterministically or flakily. In this question we look at this data from another angle. Table 7 breaks out the data by each subject as follows. It shows, those abstract tests, followed by concrete tests in parentheses, that are deterministic only, flaky only, or mixed. All data is taken at 100 simulation seconds. These are mutually exclusive categories. For instance, in subtraction zero abstract tests are always deterministic, but 89 concrete test case are. Again zero abstract tests are flaky, while 40 concrete tests are always flaky. Last, all of the abstract tests are always mixed (some flaky, some deterministic), with 224 concrete test cases falling into this category. We also can determine from this data that 353 concrete tests failed at least once in this study. For the majority of test cases, flakiness is dependent on the mutant that is being tested. The second line of this table shows that for subtraction, 3 mutants are always found deterministically, none are always flaky, and that 6 have a mixed behavior. There is only one mutant, in Hailstone, that is always flaky. In approximate majority, none of the mutants are always deterministic and/or always flaky.

Table 7: No. of Deterministic, Flaky and Mixed test cases and mutants by subject, Subtraction (S), Hailstone (H) and Approximate Majority (AM). Tests are listed as both Abstract (Abs) and Concrete (Conc.), in parentheses.

Subject	Type	Determ. Abs(Conc)	Flaky Abs(Conc)	Mixed Abs(Conc)
S	Test	0 (89)	0 (40)	9(224)
	Mutant	3	0	6
H	Test	0(0)	0(0)	7(34)
	Mutant	5	1	4
AM	Test	0(49)	8(1911)	16(877)
	Mutant	0	0	9

We now return to the time data to see how determinism and flakiness impacts fault detection over time. We use the subtraction and hailstone data. Figure 4 shows subtraction (top) and hailstone by time for deterministic only faults and flaky only faults. These graphs are the same as those from RQ2, but split out by category. In subtraction the number of tests failing deterministically increases over time, while the number failing flakily decreases. In hailstone, we see an initially higher deterministic set of failing tests, followed by a decrease and finally we see an increase as the CRN stabilizes.

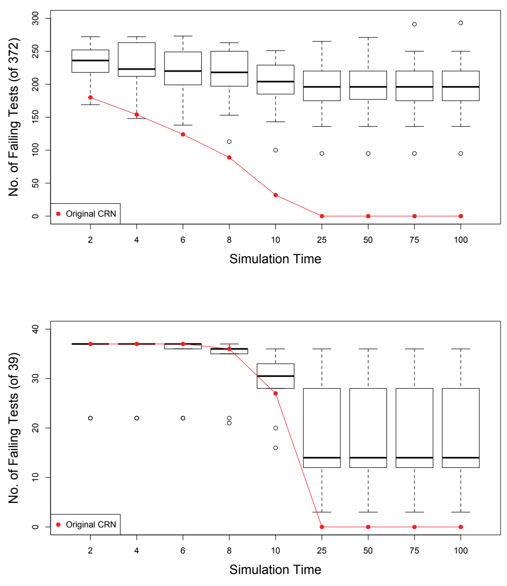


Figure 3: Subtraction, Hailstone, Faults over time.

The flaky tests also reduce over time. This suggests that some of the flakiness we see is due to simulation time. While we believe that we ran our CRNs long enough (100 simulation seconds) for all to stabilize, we can't, of course be sure. We do believe that the reasons for flakiness go beyond time. We describe some examples of flaky tests/mutants in our discussion section.

Summary of RQ3. We conclude that stochasticness plays a big role in testing CRNs. We have built in iterations (100 repetitions for this study).

5.4 Discussion

In this section we investigate CRN mutation behavior in more detail using two of the CRNs from this study.

Hailstone H5 Mutation removes the reaction $PE + PE \longrightarrow PE$ reaction from the CRN. This is an interesting mutation since it has no effect on the functional behavior, but violates the specification. This is directly tested by the CRN oracle property $FG[PO + PE == 1]$ which says that eventually there is always only one of PO or PE molecule in the system, however, the mutation allows multiple PE molecules to accumulate when the CRN terminates. This accumulation of PE molecules only occurs on even inputs.

Even when the input is even and the accumulation of PE molecules could happen, the order of reactions that fire may mask the error in the mutation. For example, if all the firings of the reactions $PO + PE \longrightarrow PO$ and $PO + PO \longrightarrow PE$ alternate, the numbers of PE molecules when the CRN stabilizes will be 1, even though there is no $PE + PE \longrightarrow PE$ reaction possible. However, a different sequence of reactions can result in multiple PE molecules when the CRN terminates. The hailstone H5 mutation is never detected by odd inputs, and only detected less than 50% of the time on even inputs, and only on a single test that utilized internal species (internal test).

In addition, this CRN is designed and constructed using CRN subcomponents, e.g., the three reactions that compute parity control a multiplexer to decide if the output is $3N + 1$ or $N/2$. While the parity CRN was flawed but had no effect on the functional output, this is not always the case. If the parity CRN is utilized in other systems where it is critical that only a single PE or PO molecule be present at the end of the computation, this other system would fail.

Approximate Majority A3 Mutation highlights the probabilistic nature of this CRN, how this manifests as flakiness, and how it can fool oracles. The first reaction is mutated so the species $X1$ is replaced by the species U. The effect of this on the overall system gives a slight unfair advantage to species $X2$. Depending on the input, this can fool hyper tests. Abstract Test 3 on with concrete input $X1 = 11$ and $X2 = 12$ only fails twice out of 100 runs. Abstract test 3 states that if $X2$ has an advantage ($X2 > X1$) then $X2$ should win. Compare this result with that of the correct AM CRN where it fails 49 times on the same input. The reason is that the AM CRN on equal inputs should yield "$X1$ wins" 50 percent of the time, and as the difference between $X1$ and $X2$ increases with $X2 > X1$ this frequency decreases. With this mutation, $X2$ is helped, and thus returns the correct majority species with better frequency.

5.5 A Roadmap for the Future of CRN Testing

In this paper we have presented an initial framework for CRN testing. We have observed many interesting future directions that we summarize briefly here.

Automated Specifications and Test Cases. We manually created the specifications and test cases for our CRNs. We see many opportunities for automated generation (both partial and complete) from the CRN models.

CRN Flakiness. As demonstrated, test flakiness is an inherent part of ChemTest. We ran all tests 100 times in our experiments to ensure our results were valid. However, we believe that it is possible to determine a sufficient number of iterations for testing. The topic of flakiness and its relation to flakiness in traditional environments is an open and interesting question.

Mutation Operators. In this work we used mutation testing to evaluate the quality of our test cases. Recent research on concurrent and flaky mutation testing [26, 57] suggests that mutation testing should be customized for this new environment. While we have seen some interesting faults that are similar to those which we have observed in our own programs, a set of sufficient mutation operators and a theory of mutation testing for CRNs is needed.

Performance Optimization. In this work our focus was on correctness, however, some of the oracle evaluation was resource intensive. Better algorithms to improve this aspect of ChemTest, including the evaluation of partial traces and states of the LTL operators, are needed.

Simulation Parameters. Several of our simulation parameters were chosen based on simple heuristics. The best threshold for considering parameters such as simulation time, with respect to the input size or other characteristics of the CRN warrants further investigation. These may be determined by both theoretical analysis and experimental tuning.

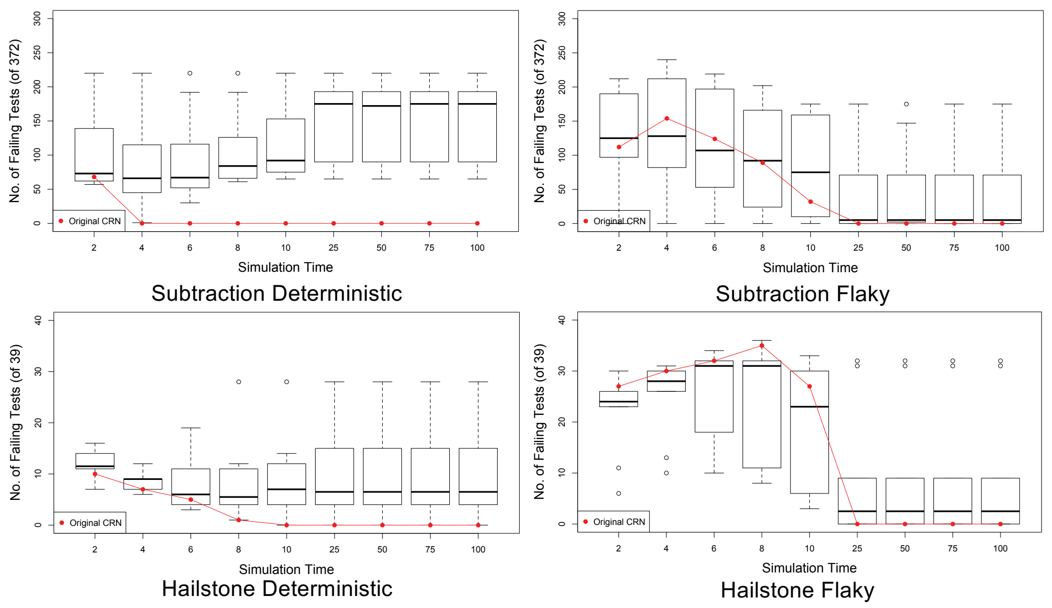


Figure 4: Faults found over time (deterministic versus flaky)

6 RELATED WORK

There has been considerable research on defining and programming CRNs for various tasks [5, 24, 27] including the development of languages that can be compiled down to CRNs [63]. We focus primarily on validating the correctness of CRNs. The state of the art is to use model checking or automated theorem proving [21, 22, 28, 34, 36, 39]. CRNs can be modeled as deterministic (using systems of differential equations), concurrent and probabilistic (using continuous time Markov model) systems [4, 25]. We focus on stochastic CRNs which are both concurrent and probabilistic. There is a body of work in concurrent testing [10, 45, 55, 62]. We don't attempt to reference it all here since it is geared towards traditional coding constructs. Some work analyzing concurrency at the nanoscale using CRNs has also been investigated (see [37] for example). In our work, the physical properties of the model drive the testing and do not explicitly change the order of firing reactions. While locking mechanisms can be achieved by program design in CRNs, it is important to note that these systems are themselves CRNs; the underlying physics in these systems are not changed. The CRN model is also related to the standard Petri net model which is widely studied [9]. However, the CRN model which defines a continuous time Markov model requires embellishments to the Petri net model. Petri nets can be used to automate test generation, but we do not explore that here. Instead we utilize our own LTL-like temporal logic which is natural for the expression of test oracles.

There has been research on test flakiness (see [7, 38, 41, 57] as a sample), however, much of that work focuses on programming constructs in traditional programming languages. We use the same notion of flakiness, but we explicitly expect and support this phenomenon. Last, there has been research on probabilistic programming [1, 19, 20, 45]. Some CRNs are probabilistic and ChemTest supports that construct, but is not specifically about solving probabilistic programming problems.

7 CONCLUSIONS AND FUTURE WORK

We presented ChemTest an end to end testing framework for chemical reaction networks. ChemTest formalizes test requirements in an LTL-like language and uses this to specify constraints on the inputs and abstract tests. It supports functional, metamorphic, internal and hyper tests. Simulations are run multiple times to handle flakiness. In a case study we see on average that functional tests find 66.5% of the mutants, while metamorphic test find 80.4%. The internal and hyper tests find 65.4% and 53.6% respectively. In addition, time of evaluation impacts fault detection. None of our abstract tests are fully deterministic and 21% are flaky across all concrete test inputs. In future work we plan to apply ChemTest to more complex CRNs such as those which require all metamorphic tests, integration across multiple CRN units and other probabilistic programs. We also will optimize the oracle processing which was a bottleneck in this study and build ChemTest directly into MatLab.

8 ACKNOWLEDGEMENTS

We thank the reviewers for their insightful comments. This work is supported in part by NSF Grant numbers CCF-1909688, CCF-1901543, and FET-1900716.

REFERENCES

[1] Aws Albarghouthi, Loris D'Antoni, Samuel Drews, and Aditya V. Nori. 2017. FairSquare: Probabilistic Verification of Program Fairness. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 80 (Oct. 2017), 30 pages. https://doi.org/10.1145/3133904
[2] Dan Alistarh, Bartłomiej Dudek, Adrian Kosowski, David Soloveichik, and Przemysław Uznański. 2017. Robust Detection in Leak-Prone Population Protocols. (09 2017), 155–171.
[3] J. H. Andrews, L. C. Briand, and Y. Labiche. 2005. Is Mutation an Appropriate Tool for Testing Experiments?. In *Proceedings of the 27th International Conference*

on Software Engineering (St. Louis, MO, USA) *(ICSE '05)*. Association for Computing Machinery, New York, NY, USA, 402–411. https://doi.org/10.1145/1062455.1062530

[4] Rutherford Aris. 1965. Prolegomena to the rational analysis of systems of chemical reactions. *Archive for Rational Mechanics and Analysis* 19, 2 (1965), 81–99.

[5] Stefan Badelt, Seung Woo Shin, Robert F. Johnson, Qing Dong, Chris Thachuk, and Erik Winfree. 2017. A General-Purpose CRN-to-DSD Compiler with Formal Verification, Optimization, and Simulation Capabilities. In *Proceedings of the 23rd International Conference on DNA Computing and Molecular Programming (Lecture Notes in Computer Science)*. 232–248. https://doi.org/10.1007/978-3-319-66799-7_15

[6] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press. https://doi.org/10.5555/1373322

[7] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: Automatically Detecting Flaky Tests. In *ICSE*. Association for Computing Machinery, New York, NY, USA, 433–444. https://doi.org/10.1145/3180155.3180164

[8] Michael A. Boemo, Alexandra E. Lucas, Andrew J. Turberfield, and Luca Cardelli. 2016. The Formal Language and Design Principles of Autonomous DNA Walker Circuits. *ACS Synthetic Biology* 5, 8 (2016), 878–884. https://doi.org/10.1021/acssynbio.5b00275

[9] W. Brauer, W. Reisig, and G. Rozenberg (Eds.). 1987. *Petri Nets: Applications and Relationships to Other Models of Concurrency*. Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-17906-2

[10] Yan Cai and Zijiang Yang. 2016. Radius Aware Probabilistic Testing of Deadlocks with Guarantees. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE 2016)*. Association for Computing Machinery, New York, NY, USA, 356–367. https://doi.org/10.1145/2970276.2970307

[11] Ho-Lin Chen, David Doty, and David Soloveichik. 2014. Deterministic function computation with chemical reaction networks. *Natural Computing* 13, 4 (01 Dec 2014), 517–534. https://doi.org/10.1007/s11047-013-9393-6

[12] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. 2018. Metamorphic Testing: A Review of Challenges and Opportunities. *ACM Comput. Surv.* 51, 1, Article 4 (Jan. 2018), 27 pages. https://doi.org/10.1145/3143561

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

[14] Yuan-Jyue Chen, Benjamin Groves, Richard A. Muscat, and Georg Seelig. 2015. DNA nanotechnology from the test tube to the cell. *Nature Nanotechnology* 10 (2015), 748–760. https://doi.org/10.1038/nnano.2015.195

[15] Kevin Cherry and Lulu Qian. 2018. Scaling up molecular pattern recognition with DNA-based winner-take-all neural networks. *Nature* 559 (2018), 370–376. Issue 7714. https://doi.org/10.1038/s41586-018-0289-6

[16] Anne Condon, Monir Hajiaghayi, David G. Kirkpatrick, and Ján Manuch. 2017. Simplifying Analyses of Chemical Reaction Networks for Approximate Majority. In *Proceedings of the 23rd International Conference on DNA Computing and Molecular Programming (Lecture Notes in Computer Science, Vol. 10467)*. Springer, 188–209. https://doi.org/10.1007/978-3-319-66799-7_13

[17] David Doty, Jack H Lutz, Matthew J Patitz, Robert T Schweller, Scott M Summers, and Damien Woods. 2012. The tile assembly model is intrinsically universal. In *Proceedings of the 53rd Symposium on Foundations of Computer Science*. IEEE, 302–310. https://doi.org/10.1109/FOCS.2012.76

[18] David Doty and Matthew J. Patitz. 2009. A Domain-Specific Language for Programming in the Tile Assembly Model. In *Proceedings of the 15th International Conference on DNA Computing and Molecular Programming (Lecture Notes in Computer Science, Vol. 5877)*, Russell Deaton and Akira Suyama (Eds.). Springer, 25–34. https://doi.org/10.1007/978-3-642-10604-0_3

[19] Saikat Dutta, Owolabi Legunsen, Zixin Huang, and Sasa Misailovic. 2018. Testing Probabilistic Programming Systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 574–586. https://doi.org/10.1145/3236024.3236057

[20] Saikat Dutta, Wenxian Zhang, Zixin Huang, and Sasa Misailovic. 2019. Storm: Program Reduction for Testing and Debugging Probabilistic Programming Systems. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 729–739. https://doi.org/10.1145/3338906.3338972

[21] S. Ellis, E. Henderson, Titus H. Klinge, James I. Lathrop, J. Lutz, R. Lutz, Divita Mathur, and A. Miner. 2014. Automated requirements analysis for a molecular watchdog timer. In *ASE '14*. Association for Computing Machinery, New York, NY, USA, 767–778. https://doi.org/10.1145/2642937.2643007

[22] Samuel J. Ellis, Titus H. Klinge, James I. Lathrop, Jack H. Lutz, Robyn R. Lutz, Andrew S. Miner, and Hugh D. Potter. 2019. Runtime Fault Detection in Programmed Molecular Systems. *ACM TOSEM* 28 (2019), 6–20. https://doi.org/10.1145/3295740

[23] Samuel J. Ellis, James I. Lathrop, and Robyn R. Lutz. 2017. State logging in chemical reaction networks. In *Proceedings of the 4th ACM International Conference on Nanoscale Computing and Communication, NANOCOM 2017, Washington, DC, USA, September 27-29, 2017*. Association for Computing Machinery, New York, NY, USA, 23:1–23:6. https://doi.org/10.1145/3109453.3109456

[24] Martin Feinberg. 1979. Lectures On Chemical Reaction Networks. http://www.crnt.osu.edu/LecturesOnReactionNetworks.

[25] Daniel T Gillespie. 2009. The Deterministic Limit of Stochastic Chemical Kinetics. *The Journal of Physical Chemistry B* 113, 6 (2009), 1640–1644. https://doi.org/10.1021/jp806431b

[26] Milos Gligoric, Lingming Zhang, Cristiano Pereira, and Gilles Pokam. 2013. Selective Mutation Testing for Concurrent Code. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013)*. Association for Computing Machinery, New York, NY, USA, 224–234. https://doi.org/10.1145/2483760.2483773

[27] Jeremy Gunawardena. 2003. Chemical Reaction Network Theory for in-silico Biologists. http://www.jeremy-gunawardena.com/papers/crnt.pdf.

[28] Arie Gurfinkel, Marsha Chechik, and Benet Devereux. 2003. Temporal Logic Query Checking: A Tool for Model Exploration. *IEEE Transactions on Software Engineering* 29, 10 (2003), 898–914. https://doi.org/10.1109/TSE.2003.1237171

[29] Dongran Han, Suchetan Pal, Jeanette Nangreave, Zhengtao Deng, Yan Liu, and Hao Yan. 2011. DNA Origami with Complex Curvatures in Three-Dimensional Space. *Science* 332, 6027 (2011), 342–346.

[30] Yonggang Ke, Luvena L. Ong, William M. Shih, and Peng Yin. 2012. Three-Dimensional Structures Self-Assembled from DNA Bricks. *Science* 338, 6111 (2012), 1177–1183.

[31] Titus H. Klinge. 2016. Robust Signal Restoration in Chemical Reaction Networks. In *Proceedings of the 3rd International Conference on Nanoscale Computing and Communication*. ACM, New York, NY, USA, 6:1–6:6. https://doi.org/10.1145/2967446.2967465

[32] Titus H. Klinge, James I. Lathrop, and Jack H. Lutz. 2020. Robust biomolecular finite automata. *Theoretical Computer Science* 816, C (2020). https://doi.org/10.1016/j.tcs.2020.01.008

[33] M. Kwiatkowska, G. Norman, and D. Parker. 2011. PRISM 4.0: Verification of Probabilistic Real-time Systems. In *Proc. 23rd International Conference on Computer Aided Verification (CAV'11) (LNCS, Vol. 6806)*, G. Gopalakrishnan and S. Qadeer (Eds.). Springer, 585–591. https://doi.org/10.1007/978-3-642-22110-1_47

[34] Marta Kwiatkowska and Chris Thachuk. 2014. Probabilistic model checking for biology. *Software Systems Safety* 36 (2014), 165–189.

[35] Jefferey C. Lafarias (Ed.). 2010. *The ultimate challenge : the 3x + 1 problem*. American Mathematical Society.

[36] Matthew R. Lakin, David Parker, Luca Cardelli, Marta Kwiatkowska, and Andrew Phillips. 2012. Design and analysis of DNA strand displacement devices using probabilistic model checking. *Journal of the Royal Society Interface* 9, 72 (2012), 1470–1485. https://doi.org/10.1098/rsif.2011.0800

[37] Matthew R. Lakin, Darko Stefanovic, and Andrew Phillips. 2016. Modular verification of chemical reaction network encodings via serializability analysis. *Theoretical Computer Science* 632 (2016), 21 – 42. https://doi.org/10.1016/j.tcs.2015.06.033 Verification of Engineered Molecular Devices and Programs.

[38] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. 2019. IDFlakies: A framework for detecting and partially classifying flaky tests. In *Proceedings - 2019 IEEE 12th International Conference on Software Testing, Verification and Validation, ICST 2019*. 312–322. https://doi.org/10.1109/ICST.2019.00038

[39] James I. Lathrop, Jack H. Lutz, Robyn R. Lutz, Hugh D. Potter, and Matthew R. Riley. 2020. Population-induced phase transitions and the verification of chemical reaction networks. In *Proceedings of the 26th International Conference on DNA Computing and Molecular Programming (DNA 2020)*. To appear.

[40] Anders Lundgren and Upulee Kanewala. 2016. Experiences of testing bioinformatics programs for detecting subtle faults. In *Proceedings of the International Workshop on Software Engineering for Science - SE4Science '16*. 16–22.

[41] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An Empirical Analysis of Flaky Tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 643–653. https://doi.org/10.1145/2635868.2635920

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

[43] Akbar Siami Namin and Sahitya Kakarla. 2011. The Use of Mutation in Testing Experiments and Its Sensitivity to External Threats. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11)*. Association for Computing Machinery, New York, NY, USA, 342–352. https://doi.org/10.1145/2001420.2001461

[44] T. J. Ostrand and M. J. Balcer. 1988. The category-partition method for specifying and generating functional tests. *Commun. ACM* 31 (1988), 678–686.

[45] Burcu Kulahcioglu Ozkan, Rupak Majumdar, Filip Niksic, Mitra Tabaei Befrouei, and Georg Weissenbacher. 2018. Randomized Testing of Distributed Systems with Probabilistic Guarantees. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 160 (Oct. 2018), 28 pages. https://doi.org/10.1145/3276530

[46] Michael Pedersen and Andrew Phillips. 2009. Towards programming languages for genetic engineering of living cells. *Journal of the Royal Society Interface* 6 (April 2009), S437–S450. https://doi.org/10.1098/rsif.2008.0516.focus

[47] Andrew Phillips and Luca Cardelli. 2009. A programming language for composable DNA circuits. *Journal of the Royal Society Interface* 6, 4 (2009), S419–S436.

[48] Lulu Qian and Erik Winfree. 2011. Scaling up digital circuit computation with DNA strand displacement cascades. *Science* 332, 6034 (2011), 1196–1201.

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

[50] Lulu Qian, Erik Winfree, and Jehoshua Bruck. 2011. Neural network computation with DNA strand displacement cascades. *Nature* 475, 7356 (2011), 368–372.

[51] Paul W. K. Rothemund. 2006. Folding DNA to create nanoscale shapes and patterns. *Nature* 440, 7082 (2006), 297–302.

[52] Christian E. Schafmeister. 2016. CANDO: A Compiled Programming Language for Computer-Aided Nanomaterial Design and Optimization Based on Clasp Common Lisp. In *Proceedings of the 9th European Lisp Symposium on European Lisp Symposium* (Kraków, Poland). European Lisp Scientific Activities Association, 9:75–9:82. https://doi.org/10.5555/3005729.3005738

[53] Nicholas Schiefer and Erik Winfree. 2015. Universal Computation and Optimal Construction in the Chemical Reaction Network-Controlled Tile Assembly Model. In *Proceedings of the 21st International Conference on DNA Computing and Molecular Programming*. Springer, 34–54.

[54] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés. 2016. A Survey on Metamorphic Testing. *IEEE TSE* 42, 9 (Sept 2016), 805–824. https://doi.org/10.1109/TSE.2016.2532875

[55] Koushik Sen. 2007. Effective random testing of concurrent programs. In *ASE'07 - 2007 ACM/IEEE International Conference on Automated Software Engineering*. Association for Computing Machinery, New York, NY, USA, 323–332. https://doi.org/10.1145/1321631.1321679

[56] Ehud Shapiro and Yaakov Benenson. 2006. Bringing DNA computers to life. *Scientific American* 294, 5 (2006), 44–51.

[57] August Shi, Jonathan Bell, and Darko Marinov. 2019. Mitigating the Effects of Flaky Tests on Mutation Testing. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 112–122. https://doi.org/10.1145/3293882.3330568

[58] Lloyd M. Smith. 2010. Nanotechnology: Molecular robots on the move. *Nature* 465, 7295 (2010), 167–168.

[59] David Soloveichik, Matthew Cook, and Erik Winfree. 2008. Combining self-healing and proofreading in self-assembly. *Natural Computing* 7, 2 (2008), 203–218. https://doi.org/10.1007/s11047-007-9036-x

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

[61] David Soloveichik, Georg Seelig, and Erik Winfree. 2009. DNA as a Universal Substrate for Chemical Kinetics. In *DNA Computing (Lecture Notes in Computer Science, Vol. 5347)*. 57–69. https://doi.org/10.1007/978-3-642-03076-5_6

[62] Valerio Terragni and Mauro Pezzè. 2018. Effectiveness and Challenges in Generating Concurrent Tests for Thread-Safe Classes. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. Association for Computing Machinery, New York, NY, USA, 64–75. https://doi.org/10.1145/3238147.3238224

[63] Marko Vasic, David Soloveichik, and Sarfraz Khurshid. 2018. CRN++: Molecular Programming Language. In *DNA Computing and Molecular Programming*, David Doty and Hendrik Dietz (Eds.). Springer International Publishing, 1–18.

[64] Daniel Wilhelm, Jehoshua Bruck, and Lulu Qian. 2018. Probabilistic switching circuits in DNA. *Proceedings of the National Academy of Sciences* 115 (01 2018), 201715926. https://doi.org/10.1073/pnas.1715926115