Inference and Test Generation Using Program Invariants in Chemical Reaction Networks

Michael C. Gerten
Iowa State University
Ames, Iowa, USA
mcgerten@iastate.edu

Alexis L. Marsh
Iowa State University
Ames, Iowa, USA
almarsh@iastate.edu

James I. Lathrop
Iowa State University
Ames, Iowa, USA
jil@iastate.edu

Myra B. Cohen
Iowa State University
Ames, Iowa, USA
mcohen@iastate.edu

Andrew S. Miner
Iowa State University
Ames, Iowa, USA
asminer@iastate.edu

Titus H. Klinge
Iowa State University
Ames, Iowa, USA
tklinge@iastate.edu

ABSTRACT

Chemical reaction networks (CRNs) are an emerging distributed computational paradigm where programs are encoded as a set of abstract chemical reactions. CRNs can be compiled into DNA strands which perform the computations in vitro, creating a foundation for intelligent nanodevices. Recent research proposed a software testing framework for stochastic CRN programs in simulation, however, it relies on existing program specifications. In practice, specifications are often lacking and when they do exist, transforming them into test cases is time-intensive and can be error prone. In this work, we propose an inference technique called ChemFlow which extracts 3 types of invariants from an existing CRN model. The extracted invariants can then be used for test generation or model validation against program implementations. We applied ChemFlow to 13 CRN programs ranging from toy examples to real biological models with hundreds of reactions. We find that the invariants provide strong fault detection and often exhibit less flakiness than specification derived tests. In the biological models we showed invariants to developers and they confirmed that some of these point to parts of the model that are biologically incorrect or incomplete suggesting we may be able to use ChemFlow to improve model quality.

CCS CONCEPTS

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

KEYWORDS

chemical reaction networks, test generation, invariants, Petri nets

ACM Reference Format:
Michael C. Gerten, Alexis L. Marsh, James I. Lathrop, Myra B. Cohen, Andrew S. Miner, and Titus H. Klinge. 2022. Inference and Test Generation Using Program Invariants in Chemical Reaction Networks. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510176

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05.
https://doi.org/10.1145/3510003.3510176

1 INTRODUCTION

Many non-traditional programming paradigms have recently emerged, including techniques to leverage physical processes to embed computations in nanodevices (called *molecular programming*) [13, 45, 49, 53]. Chemical reaction networks (CRNs) are one such paradigm where programs are encoded as a set of abstract chemical reactions [9, 10, 15, 54]. CRNs are of special interest because they can be compiled into a program using DNA strands and then executed in vitro [4, 51]. As a result, CRNs help catalyze the development of intelligent nanodevices; devices that could be leveraged for purposes such as intelligent drug delivery [1, 17, 26].

Since CRNs form the foundation of living organisms, these are also of interest to biochemists and synthetic biologists (engineers of new biological functions), who build models to describe the physical *program implementations* (both synthesized and natural). There are now numerous repositories of reusable biological parts available online, many of which can be represented as CRNs [8, 45]. Some of these repositories have even been likened to the Github for synthetic biology [8, 11, 23]. Several recent advances include higher level programming languages that compile into CRNs [48, 54]. Recently CRNs have even been formulated to represent neural networks and these have been shown to be effective and efficient at common classification tasks [49, 53] suggesting a potential for nanolearning. If we examine recent publications promoting novel uses/programming approaches for CRNs we find that as many as half of the papers in the recent (2020/21) DNA conferences [24] are on this topic. And if we examine the ACM digital library we find recent papers crossing domains such as software engineering (in venues such as TOSEM, TSE, ICSE, ASE, RE) and computational biology, networking and chemistry domains.

While the science of programming CRNs is exploding, validating their correctness lags. Much of the state of the art in validating CRNs uses model checking or other formal approaches such as theorem proving [18, 33, 36]. However, scalability of model checking is often an issue, in particular in CRNs that have faults [20, 32, 36]. Users have instead started to ask about testing on forums such as MATLAB help forum [39].

The CRN community uses two primary semantics for CRNs (which differ in how the CRNs are simulated/evaluated). In this work we focus on one type, stochastic CRNs. In recent work we proposed an automated software testing framework for stochastic CRNs, ChemTest [20]. However, ChemTest relies on the existence

of program specifications. These specifications define both the program inputs and the oracles using a temporal logic.

As we demonstrate in our motivating examples, writing manual specifications for stochastic CRNs is time consuming and potentially error prone. And as in traditional software, CRNs may have ambiguous specifications, may be autogenerated, or may come without any specifications (such as biological organisms). All of these issues suggest the need for automated inference and generation techniques. Since CRNs are known to be equivalent to a common structure in distributed software testing, Petri nets [9], and there is a large body of literature on ways to analyze these networks [41], we ask if we can leverage that research to analyze CRNs and extract (invariant) properties of the CRNs. If we can infer a CRNs behavior sufficiently, we have a powerful technique that can be used for test generation, and to validate known models.

In this paper, we explore this idea. We have created a technique called *ChemFlow*. ChemFlow extracts linear invariants from stochastic CRNs using a form of Gaussian elimination. It creates different types of invariants, some which can be easily incorporated into a testing framework. Others require more complex harnessing (reaction counting). We implemented and evaluated both in this work. ChemFlow is significant for the software engineering community; as we move to the new nanocomputational world (as is already happening) we need to provide strong software engineering practices that can guide and ensure program correctness.

We evaluate ChemFlow by first comparing tests created from its invariants against manually generated test cases for a wide range of CRN programs which have specifications. We find that the invariants have good fault detection and that they tend to be less flaky. We also see that adding reaction counting increases the quality of the invariants. We then evaluate the use of invariant detection in four subjects which lack manual specifications. Two of these are based on metabolic models of a living organism *Escherichia coli*. We presented the biological invariants to developers (bioinformaticists) who work with these types of models and write programs to analyze them. They pointed out that some invariants represent potential inconsistencies between the model and known behavior.

The contributions of this work are:

(1) A linear invariant inference technique for chemical reaction networks, called ChemFlow;
(2) A large study showing invariants provide good fault detection and can indicate potential problems with model quality;
(3) An analysis of some tradeoffs between different types of invariants and manual specifications.

In the next section we present some motivating examples and background. We follow this with a presentation of our invariant test generation technique (Section 3). We then present our study in Sections 4 and 5, followed by discussion. We present related work in Section 6. Finally, we conclude and present future work.

2 MOTIVATION AND BACKGROUND

We begin with some motivating examples to demonstrate the need for automated invariant inference in CRNs. We then present some background on CRNs and the state of the art in CRN testing.

2.1 Three Motivating Examples

Scenario One: Unclear Specifications We begin with one of our subject programs obtained from a tutorial on building CRNs [9] for which we needed to write test cases. The author of this tutorial presents a CRN that satisfies the specification *Suppose we are given a state with an unknown number of molecules of species X and Y and we want to decide whether or not #X is equal to #Y modulo 3*. They then present a program with five reactions (see background below for specifics of CRN program notation etc.). Our interpretation of this specification is that the program accepts two inputs (X and Y) and performs modulo 3 arithmetic. It then returns a Boolean result (0 or 1) stating if these are equal. We quickly realized the specification was unclear. We initially assumed a 0 or 1 return value, but of course, this may also be 0 or any strictly positive value (if true). The specification fails to state which way the program works. In our initial (incorrect) test cases we checked for a return value of 1 as true. Many tests failed and it took hours and many iterations to determine the exact specifications for this simple, correct CRN.
Scenario Two: Autogenerated Code. Another of our study subjects (Hailstone 4), is an auto-generated CRN created by a compiler using a high level domain specific language. This function (Hailstone) is a well known function. However, the particular formulation of the program has 91 reactions and 50 species. Although we know the general functional properties of Hailstone, we do not know the exact values of the various input/outputs and we could not extract complete specifications manually for this program.
Scenario Three: Model Quality for Existing Programs. The last example is for evaluating conformance of a model and its implementation. We again look at one of our study subjects, a CRN extracted from the metabolic model of a living organism. The CRN is a model, not the actual program implementation. While our organism is well studied, there are no exact specifications for a living organism. These are learned over time using years of collective experimentation. In addition, the organisms evolve. We, therefore, want to extract specifications from the existing model and compare with the organism's known behaviors. The biological CRN has almost 500 reactions and as many species.

2.2 Chemical Reaction Network Programs

Chemical reaction networks (CRNs) and its many variants are widely used to model the interactions of molecules [15, 19]. Here, we use the stochastic mass-action variant which models the state of the network with integral count of molecules rather than real-valued concentrations. CRNs are now commonly used to develop chemical algorithms by carefully specifying the interactions between abstract molecules. Recently, researchers have developed methods to synthesize abstract chemical networks into strands of DNA molecules [4, 13, 51].

Roughly, the stochastic model of a CRN is defined as a pair $N = (S, R)$ where S is a finite set of species (abstract molecule types) and R is a finite set of *reactions* that operate over the species. A reaction is composed of reactants and products, which are vectors of species, and are often denoted using the format

$$A + B \xrightarrow{2.43} C + D.$$

The reaction above depicts the interaction of two molecules, A and B, that come together in solution, interact, and produce molecules C and D. The rate of the reaction is proportional to the rate constant 2.43 and the product of the number of A and B molecules in solution. In the stochastic model, this overall rate determines a probability that the reaction will occur and how long it will remain in the new state (sojourn time). Thus, if the number of A molecules is 10 and the numbers of B molecules is 2, then the propensity of this reaction firing at a specific time is determined by the quantity $10 \cdot 2 \cdot 2.43/V$ where V is the volume of the solution. The probability that the reaction fires is determined by its propensity and the propensity of all the other reactions in the system. When the volume is constant we can write the above reaction as $A + B \longrightarrow C + D$.

We give an example CRN to illustrate the semantics of a stochastic CRN. Consider the CRN $N = (S, R)$ with species $S = \{X, Y, PE, PO\}$ and R consisting of the following four reactions.

$$X \longrightarrow PO + Y \quad \text{(R1)}$$

$$PO + PO \longrightarrow PE \quad \text{(R2)}$$

$$PE + PO \longrightarrow PO \quad \text{(R3)}$$

$$PE + PE \longrightarrow PE \quad \text{(R4)}$$

This CRN computes *parity*, i.e., whether the number of input molecules X is even or odd. The output is a single molecule: PO (odd number in X) or PE (even number in X), and preserves the initial number of X in the species Y. To see how this works, we initialize X to 5 and PE, PO, and Y to zero, noting that the number of molecules in X is an odd number. At the end of the computation, the species PE should be 0 and PO should contain a single molecule that indicates the initial input in X was odd. We can examine the behavior more clearly by looking at a *trace* of the CRN over time. Table 1 shows the trace and enumerates the molecule counts of the CRN after each reaction occurs. It represents only one of many different orderings the reactions may occur. Note that if not all orders lead to the correct results, the failure is probabilistic, which leads to flaky CRNs and flaky CRN testing. Flaky tests can also occur when they are evaluated before the CRN stabilizes.

Table 1: Example of a CRN execution

Time	Reaction	X	PO	PE	Y
0	Initial Values	5	0	0	0
0.0410	$X \rightarrow PO + Y$	4	1	0	1
0.1556	$X \rightarrow PO + Y$	3	2	0	2
0.1773	$PO + PO \rightarrow PE$	3	0	1	2
0.3567	$X \rightarrow PO + Y$	2	1	1	3
0.7181	$X \rightarrow PO + Y$	1	2	1	4
0.7622	$PO + PO \rightarrow PE$	1	0	2	4
0.7828	$PE + PE \rightarrow PE$	1	0	1	4
1.5815	$X \rightarrow PO + Y$	0	1	1	5
1.9690	$PE + PO \rightarrow PO$	0	1	0	5
10	End of simulation	0	1	0	5

As we see in Table 1, at time 0.0410 reaction (R1) fires, which converts a molecule of X into a molecule of PO. Since there is only a single PO molecule present, no other reactions can fire, and so reaction (R1) must fire again. This occurs at time 0.1156 where another X is converted and we now have two PO molecules. At this point, both reactions (R1) and (R2) are enabled, and one is chosen probabilistically proportional to the reaction's propensity. On our example, reaction (R2) is chosen at time 0.1773. As more reactions are enabled, the state of the molecules changes according to the rules of the reactions that fire, eventually leaving a single molecule of PE or PO. In this state, no reaction can fire and the system remains stable.

2.3 Testing CRNs

In prior work we proposed a stochastic CRN testing framework, *ChemTest*, which uses a multi-step process[20]. In that work we compared scalability with model checking and demonstrated the need for an alternative approach to CRN verification. We summarize ChemTest here. First, tests are formulated from specifications (as stated in our first motivating example) using a linear temporal logic (LTL). Both the program inputs and the properties which must hold (the oracles) are defined by the LTL. These become *abstract test cases*. Four types of test cases are designed. Functional test cases, are properties on the program output. Equation (1) shows a functional abstract test for subtraction from the ChemTest artifact website. It is subtracting $X2$ from $X1$ and the result goes into species Y. Y can never be negative, hence any negative values become 0. The property states if $X2$ is greater than $X1$ then *future globally* Y (at evaluation time t) will be equal to zero.

Metamorphic tests are also used. These are compared against two program executions. Equation (2) states if $X1'$ (the second execution) is greater than $X1$ and $X2$ is held constant, then *future globally* at evaluation time t, Y' is greater than Y.

$$[X2[0] > X1[0]] \rightarrow FG[Y[t] = 0] \quad (1)$$

$$[X1'[0] > X1[0]] \rightarrow FG[Y'[t] > Y[t]] \quad (2)$$

Two other tests types are defined. Internal tests can be either functional or metamorphic, but test properties of an internal species rather than a program output. Hypertests are sets of test cases across which a property holds. They are used for probabilistic programs which are not guaranteed to give the same answer each run. We do not use either of these test types in this paper.

As can be seen by the examples writing LTL test cases may be cumbersome and can require significant manual effort. It also requires a full set of program specifications.

Once abstract tests are created, ChemTest uses category partition to define sets of inputs which generate *concrete test cases*. The test specification language (TSL) [44] is a standard way to implement the category partition method on a program input space. TSL first partitions a program by its parameters and for each parameter it defines *choices* or equivalence classes of values. The set of parameters and choices for the program becomes the TSL model. In stochastic CRNs species are parameters and choices are the species concentration equivalence classes (e.g. we might use odd number of species, zero species, large number of species). Constraints between inputs can also be specified using temporal logic to help restrict invalid test inputs (some of the specifications require that $X1 > X2$ for instance). Using the concrete tests, ChemTest then runs the test cases on a CRN simulator (in the MATLAB Simbiology package [52]). The properties of each test run are checked for correctness at a

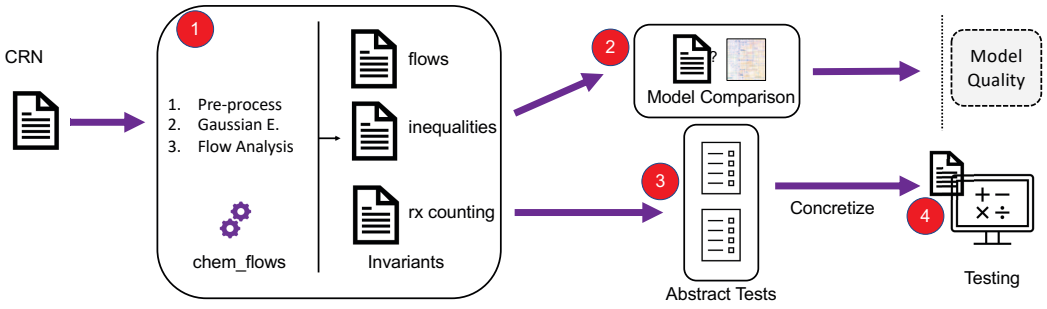

Figure 1: ChemFlow Overview. In #1 it pre-processes the data, performs Gaussian elimination and flow analysis. Once invariants are extracted they can be used either for (#2) Model comparison and quality and/or (#3) Test Generation and Testing.

specified time during simulation. Because tests can be flaky, each simulation is run a number of times.

3 CHEMFLOW

We now present ChemFlow. Figure 1 shows an overview of ChemFlow. We begin with a stochastic CRN. We then pre-process and use our tool we call `chem_flows` (Step #1) to extract invariants. The invariants can be of three types. These can then be compared against an implementation for model quality (#2) or they can be automatically translated (#3) into abstract test cases. The abstract test cases are then concretized and used for testing (#4) by assigning sets of inputs (and associated oracles) using category partition. We discuss the key aspects of invariant extraction next, followed by an example of test generation.

3.1 Invariant Extraction

Chemical reaction networks are known to be equivalent to Petri nets [9]; in fact, Petri nets were originally conceived to describe chemical processes [47], where molecules, species and reactions in CRNs become tokens, places and transitions in Petri nets. Thus, CRNs inherit most of the theoretical results originally derived for Petri nets [41] and Vector Addition Systems [29]. In particular, we utilize the well-known Petri net concept of linear invariants, or "P–flows" [50]. Each P–flow is a weighted sum of numbers of tokens in places, that is guaranteed to remain constant. For example, in the CRN specified by reactions (R1)–(R4), we see that the number of X molecules plus the number of Y molecules is a constant that depends on the initial value of X. In the trace shown in Table 1, we see that $X + Y = 5$ holds at all times during the CRN execution and that $1 \cdot X + 1 \cdot Y$ is a P–flow.

We can also derive relationships from *inequalities*. For example, $PO - Y + 2 \cdot PE \leq 0$ is an invariant of the CRN. Moreover, we can utilize relationships found in the CRN by counting the number of times a reaction fires in the CRN. This information gives us additional equality invariants. For example, in the above example $R1 + Y = 5$ says that the number of times $R1$ has fired plus the number of Y molecules is 5 at any time during the execution of the CRN. A more interesting invariant for this system is $2 \cdot R2 + PO - Y = 0$, which is also true for all time the CRN is executing. This inequality can only be evaluated with special counting harness which evaluates how often a reaction has occurred.

We now give a more detailed description of how these relationships are derived. We start with an equation for the number of molecules for each species, as the initial number of molecules modified by the number of firings of each reaction multiplied by the net change for that species when that reaction fires. For our CRN shown above, we would obtain the linear equations:

$$\begin{aligned} X &= X_{initial} - R1 \\ Y &= Y_{initial} + R1 \\ PO &= PO_{initial} + R1 - 2 \cdot R2 \\ PE &= PE_{initial} + R2 - R3 - R4. \end{aligned}$$

These hold for any non-negative number of firings for each reaction, i.e., for all time after the initial configuration. Next, we rearrange each equation, keeping the initial values on the right hand side, and moving everything else to the left hand side. We now obtain

$$\begin{aligned} X + R1 &= X_{initial} \\ Y - R1 &= Y_{initial} \\ PO - R1 + 2 \cdot R2 &= PO_{initial} \\ PE - R2 + R3 + R4 &= PE_{initial} \end{aligned}$$

We can then use a series of elementary operations, namely multiplying equations on both sides by a constant or adding one equation to another, with the goal of eliminating the reaction counting terms. For our example, adding the first equation to the second eliminates $R1$, giving us

$$X + Y = X_{initial} + Y_{initial}$$

This is the linear invariant discussed above, $X + Y = 5$, for the inputs $X_{initial} = 5$ and $Y_{initial} = 0$. Similarly, subtracting the second equation from the third gives the linear invariant

$$PO - Y + 2 \cdot R2 = PO_{initial} - Y_{initial}$$

An important observation is now in order. For any linear invariant obtained from these equations, its right-hand side, which is a constant, can be obtained from its left-hand side, by substituting the initial count for each species and zero for each reaction count. To see that this property is true, note that it holds in the starting equations, and remains true after performing any elementary operation. It therefore suffices to manipulate the left sides of the equations only, keeping in mind that the right side is always a constant that can be recovered from the initial configuration.

Based on the above discussion, we implemented a standalone tool, `chem_flows`, that reads a CRN model, and derives its linear invariants. Prior to its use, we pre-process the CRNs to a standard format and eliminate any non-integer coefficients using a multiplicative factor. We also split bi-directional reactions. `chem_flows` first indexes the CRN species and reactions, and then builds the stoichiometry matrix **M** of the CRN [9], known also as the incidence matrix in Petri net literature [50]. This matrix has a row for each species and a column for each reaction, and element $\mathbf{M}[s, r]$ is the net change in species s when reaction r occurs. The left side of the initial set of equations is given by the matrix **M** augmented with $-\mathbf{I}$, denoted as $[\mathbf{M} : -\mathbf{I}]$, where **I** is the identity matrix of dimension number of species. The augmented matrix is then reduced using Gaussian elimination with integer arithmetic. After reduction, the rows contain the linear invariants, each true at all times and for any sequence of reactions that may occur.

Our `chem_flows` tool then classifies each linear invariant into one of three categories. The first category, labeled *Flows*, contains relationships between species only, without reaction counts. These are obtained from rows whose reaction count coefficients are all zero. The second category, labeled *Inequalities*, are used to derive the inequalities used in this study, but may also be used in test harnesses that utilize reaction counting. These are obtained from rows containing at least one non-zero reaction count coefficient, and whose reaction count coefficients all have the same sign. The third category, labeled *Irreducible*, requires reaction counting to evaluate. These are obtained from rows containing reaction count coefficients of different signs, which prevent derivation of an inequality in terms of species counts.

3.2 Test Generation

We now summarize the procedure for generating test cases, using a CRN program from our study: Hailstone (H1). Step one creates invariants using the `chem_flows` tool. The Hailstone input file for the `chem_flows` program and its output is shown below:

```
X1 -> PO + H + M;
PO + PO -> PE;
PE + PO -> PO;
PE + PE -> PE;
H + H -> D;
M -> 3B + 6A;
2B + 2A ->;
PE + D -> PE + CE + Y;
PO + A -> PO + CO + Y;
CE + PO + Y -> PO + D;
CO + PE + Y -> PE + A;
```

```
Flows (== constant):
+3 X1 +3 M -1 B +1 A +1 CO
+3 H -3 M +6 D +1 B -1 A +6 Y -7 CO
+1 CE -1 Y +1 CO

Inequalities (remove reactions, <= constant):
+3 (reaction 1) -3 M +1 B -1 A -1 CO
+6 (reaction 2) +3 PO -3 M +1 B -1 A -1 CO
+6 (reaction 3) +6 (reaction 4) +3 PO -3 M +6 PE +1 B -1 A -1 CO
+1 (reaction 5) -1 D -1 Y +1 CO
+3 (reaction 6) +1 B -1 A -1 CO
+2 (reaction 7) +2 B -1 A -1 CO

Irreducible:
+1 (reaction 8) -1 (reaction 10) -1 Y +1 CO
+1 (reaction 9) -1 (reaction 11) -1 CO
```

The output from the `chem_flows` program is used to generate an abstract test (inputs and an oracle) and subsequent concrete tests. We begin by describing how we transform an expression in the *Flows* section of the output into an LTL formula.

Each expression in the flows section is transformed by setting it equal to some constant ($G[expression = consant]$) which must be true at all evaluation times of the simulation $t >= 0$. For example, in the output shown above, the first flow, `+3 X1 +3 M -1 B +1 A +1 CO`, transforms to the to LTL formula

$$+3X1 + 3M - 1B + 1A + 1CO = constant$$

To generate a concrete test we need values for the species in the formula. For the input species we use a TSL model. Once the initial values are generated, the concrete test is created by setting each of the species involved in the abstract test to its initial value. We then evaluate the expression to calculate the constant. This constant is placed within the LTL global operator G, and forms a concrete test. In the case when the species are not input species, we set them to zero. In the above example, the function has only one input species $X1$. The TSL follows ChemTest which has categories for $X1$ such as even, odd, zero or maximum and an environment which controls the maximum values (to ensure we have combinations which are small, medium and large). All the other species are set to 0. If $X1$ is set to 50, then the the concrete test

$$+3X1 + 3M - 1B + 1A + 1CO = 150$$

is used to test the output trace of the CRN. Many of our CRNs have multiple input species.

Each expression in the *Inequalities* section can be transformed into an abstract test and concrete test using a nearly identical procedure. For these expressions, any terms in the expression that refers to a reaction firing is ignored and the same procedure for flows is performed. However, the invariant is no longer an equality, but rather a less than inequality. The constant is again found by setting the species involved to their initial values and using equality to determine the constant. For example, the first expression in the inequalities section becomes

$$-3 \cdot M + B - A - CO \leq 0. \text{ (initial values of } M, B, A, CO = 0)$$

The *Irreducible* section has expressions that contain reaction firing but cannot be used as an inequality. As such, *reaction counting* must be used to utilize these expressions. The Inequalities section also contains expressions with reaction firing, hence both sections can be used with reaction counting for creating abstract and concrete tests. Since reaction counting expressions are equalities, the invariant they represent are treated the same as in the Flows section. However abstract tests from these expressions must utilize reaction counting to evaluate the property of the CRN.

3.3 Implementation

We first run the CRN models through a script that scales any reactions to eliminate non-integer coefficients. This model is then fed into the `chem_flows` tool, which extracts and classifies the invariants. A MATLAB script then processes and translates each invariant into a set of abstract test cases, based on the invariant type. The concrete inputs are generated from a TSL program. Since the invariant abstract tests have no constraints on the input values we

end up with a set of tests containing the Cartesian product of the TSL categories for all input species.

We use the ChemTest framework using MATLAB SimBiology [52] to run tests. It accepts the CRN, the set of concrete inputs and the abstract test cases. It then executes the test cases, and process the traces to extract passing or failing test cases.

4 EVALUATION

We evaluate the quality of ChemFlow by asking three research questions. Artifacts for this study are available online:[1]

RQ1: How effective are the flow invariants created by ChemFlow? We answer this question using fault detection compared with specification based test suites.

RQ2: What is the impact on effectiveness and efficiency when we add inequality and irreducible invariants? We evaluate the change in fault detection and the time to execute.

RQ3: Does ChemFlow provide useful information when we lack specifications? We examine if the invariants can find faults in the mutants and if they indicate potential model quality issues.

4.1 Study Subjects

We gathered a set of 13 benchmark CRN programs from (a) prior work on CRN testing, (b) the literature, (c) a CRN simulation tool, and (d) a metabolic model from an online database. They range in size and complexity from a single reaction to almost 500 reactions. All of the CRNs can be found on our supplementary data website. For 9 of of these CRNs, we either have existing manual test suites from ChemTest, or we followed Gerten et al.'s [20] approach to build our own test suites based on specifications. We examine the other four CRNs in a regression environment, where no manual tests are available. Table 2 summarizes the characteristics of the CRNs ordered by the number of species, followed by the number of reactions. The first column gives the CRN name and its acronym used throughout the rest of this paper. The second column references the source of the CRN and is followed by the number of species (# Species) and reactions (# React). The next two columns are the number of specifications for abstract test cases (# ATests) and the number concrete tests (# CTests). A dash indicates a lack of specifications for this subject. The next column (# Mutants) lists the number of mutant programs we used for testing (see below for details on mutants). The last column (# Unstable) indicates (with an x) which programs are considered unstable, meaning they do not have a single terminating state and are therefore likely to be either probabilistic or flaky on the correct CRNs. We summarize our programs next. The first four subjects were either obtained directly from the ChemTest artifacts [20] or are derivatives of them.

(1) **Subtraction (S)**. This CRN computes the value $X1 - X2$ and places the result in species Y.
(2) **Hailstone One (H1)**. The Hailstone function [34] has a single input species $X1$ and outputs a species Y that is $X1/2$ if $X1$ is even, and $3 \cdot X1 + 1$ if $X1$ is odd. It is composed of four separate units that compute (1) if $X1$ is even or odd, (2) a divider that computes $X1/2$, (3) a multiplier that computes $3 \cdot X1+1$, and (4) a multiplexer that selects between the output of two components based on the results of the multiplier.
(3) **Hailstone Two (H2)**. This CRN is similar to Hailstone One except it differs in the computation of $3 \cdot X1 + 1$, requiring two more reactions.
(4) **Approximate Majority (AM)**. Using four reactions, this CRN probabilistically determines whether inputs $X1$ or $X2$ has an initial majority. If $X1 > X2$, then with high probability the execution will terminate with $X1+X2$ molecules residing in $X1$ and none in $X2$. Similarly, if $X2 > X1$, then with high probability all of the molecules reside in $X2$ at termination. The probability of correct output is a function of the initial difference between $X1$ and $X2$.

The next set of subjects comes from a tutorial on CRNs [9].

(5) **Min**. This CRN computes the minimum of two input species X and Y and places the result in species Z.
(6) **Max**. This CRN computes that maximum of two input species X and Y and places the result in species Z.
(7) **XYMod3 (Mod)**. This CRN determines if two input species X and Y are congruent modulo 3. If $X \equiv Y \mod 3$, then the output species V contains at least one molecule, and otherwise it contains no molecules.

Our next subject was taken from Chen et al. [12] which contains a series of CRNs that the authors designed for optimal parallelization.

(8) **At Least One (AL1)**. This program determines if there is at least one molecule of species $A1$ and at least one molecule of $A2$ and if true, the output Y contains at least one molecule.

Our next subject is a safety critical mechanism designed for use in nanodevices, (a heartbeat detector). It has probabilistic model checking results which verify its correctness on up to 5 molecules.

(9) **Molecular Watchdog Timer (MWT)**. This program determines if a heartbeat molecule H is detected within a specified time interval (specified when designing the system). We followed Gerten et al. techniques to generate LTL properties from the original goal diagrams [18]. We then did a parameter search to find inputs for the concrete tests.

The next two programs lack specifications.

(10) **Hailstone Four (H4)**. This subject was written by some of the co-authors for a different project in a high-level, domain specific language and automatically compiled into the concrete CRN consisting of 50 species and 91 reactions[35]. Mapping the inputs and outputs from another variant of Hailstone is non-trivial.
(11) **Predator Prey Model (PP)**. This is a predator prey model taken from the online GEC CRN simulator [45] which generates CRNs that are compilable into physical systems. It follows a common predator prey model, but given the variation in how this can be implemented the lack of specifications means it is difficult to develop a sufficient set of tests.

The last two CRNs are derived from metabolic networks of well-studied living organisms. Cellular metabolism can be viewed as a distributed network of chemical reactions [22]. For this study we used the Department of Energy Systems Biology Knowledge base (KBase) [3, 30], an open science platform for biologists.

(12) ***Escherichia coli* (EC)**.
(13) ***Escherichia coli* Glucose Pathway (ECG)**.

[1] Artifact Website:https://doi.org/10.5281/zenodo.5915597

Table 2: Study Subjects. We show the subject name (and acronym), its source, the number of species (# Species), the number of reactions # React, followed by the number of manually created abstract (ATests) and concrete (CTests) tests. Last we provide the number of mutant programs and indicate if the program is unstable. Starred subjects are used in Figure 3.

Name(abbrev)	Source	# Species	# React	# ATests	# CTests	# Mutants	Unstable
Min*	Brijder[9]	3	1	12	430	23	
Subtraction (S)*	Gerten et al.[20]	3	2	9	354	32	
Approximate Majority (AM)*	Gerten et al.[20]	3	4	3	279	29	x
XYMod3 (Mod)*	Brijder [9]	3	5	5	74	24	
Max*	Brijder [9]	5	3	10	160	24	
At Least One var d. (AL1)*	Chen et al.[12]	7	5	14	262	22	x
Hailstone One (H1)*	Gerten et al.[20]	11	11	13	70	34	
Hailstone Two (H2)	Variant of Gerten et al. [20]	12	13	8	22	25	
Molecular Watchdog Timer (MWT)*	TOSEM[18]	16	22	9	27	22	x
Predator Prey (PP)	GEC[45]	27	47	—	—	23	x
E. coli Glucose Pathway (ECG)	KBase [30]	23	12	—	—	25	x
Hailstone Four (H4)	paper authors [35]	50	91	—	—	25	x
E. coli (EC)	KBase [30]	496	484	—	—	25	x

For the EC subject, we built a genome scale metabolic model of *Escherichia coli* str. K-12 substr. MG1655 on Carbon-D-Glucose using the KBase [3] Build Metabolic Model app (Version: 2.0.0). The genome was retrieved through KBase [3] from National Center for Biotechnology Information (NCBI) RefSeq Genomes [43]. Since genome annotations are incomplete, the metabolic models built from them have missing reactions which make the model unable to produce biomass (grow) using typical media. Gap-filling adds the minimal set of reactions required for the organism to produce biomass on the selected media. We used the KBase [30]'s Gapfill Metabolic Model (Version: 2.0.0) on Carbon-D-Glucose.

The ECG subject was derived from KBase model of *E. coli* glycolysis pathway. We extracted just the glycolysis pathway from the *E. coli* model. These was done by hand by one of the authors (with biological knowledge) from the complete metabolic network based on its KEGG map [28].

Once we extract the sets of invariants for the *E. coli* CRNs, we need valid input species values so the simulation is not static (i.e. the organism simulates growth). We did not attempt to add in full kinetics, which means that the growth results may not be realistic, however, we did ensure that the CRN was dynamic and compounds were being consumed throughout the simulation.

4.2 Mutations

The three subjects from ChemTest [20] include a set of mutant representing faulty CRNs. Since there is no database of faulty CRNs we follow their method to create mutant programs for the others. In other experimental settings for testing, mutants have been shown to be representative of real faults [2, 42]. Mutants are created by randomly adding or removing a reactant, product or reaction. Deletions are selected 10% of the time, while changes and additions each have a probability of 45%. Species and reactions are selected with equal probability. In the ChemTest subjects there were at most 10 mutants. We added up to 25 more in each of those subjects and up to 25 in the new subjects. Mutants which ran out of memory during our experiments (more than 32 GB) were removed.

4.3 Method

For each CRN we extract the invariants and split these into the flows, inequalities and irreducibles. For reaction counting we add instrumentation to capture reaction counts during execution. For the invariant tests, the invariants produced by running the `chem_flows` program are transformed into oracles that contain the abstract tests as described earlier. For the specification tests, the abstract tests are obtained either from existing tests (i.e. ChemTest) or hand written using expected program behavior.

For the abstract tests we use a TSL to generate concrete inputs. For specification tests we often have constraints between inputs (e.g. input one has to be greater than input two). For invariant based tests, we used the same set of TSL but remove constraints. We run each concrete test 100 times using the Simbiology simulator in MATLAB [52]. We run the original CRNs (without faults) for each subject as a baseline to confirm that the test cases are valid. For all test types, we then evaluate each of the mutant programs. We run all experiments on a server, using a homogeneous set of Intel(R) Xeon(R) Gold 6244 CPUs @ 3.60GHz with 32 gb of RAM. The *E. coli* model required additional resources, with 64 GB of RAM for the larger model. We used MATLAB R2021a-io4754x running on RedHat Enterprise Linux 7.

4.4 Independent Variables

We have two independent variables: the type of invariant (or test suite), and the testing technique. Flows are the basic invariants which represent equalities. We use these in RQ1 as a base evaluation, since they do not need any special instrumentation or translation. For RQ2 we examine if additional types of invariants (inequalities and irreducibles) impact fault detection and efficiency. For the inequalities we can evaluate these both with and without our second variable (reaction counting). For the irreducibles, we can only evaluate these using reaction counting. The instrumentation in our test harness works by adding a new species for each reaction and counting the number of times this is fired. The last type of test suite we use are the specification tests suites.

4.5 Metrics

We examine the fault detection rate (number of mutants detected over the total number of mutants) and percent of mutants which are flaky (at least one concrete test exhibits flaky behavior by passing and/or failing on at least one run and doing the opposite on another.) We measure efficiency using testing time (simulation plus oracle evaluation).

4.6 Threats to Validity

While we used a variety of CRN programs, we cannot guarantee that the results generalize beyond these programs. To lessen this threat we selected a variety of programs (stable and probabilistic) in a variety of sizes. We also provide these on our external website. A second threat is that we wrote the specification tests for subjects other than those in Gerten et al. This could bias the data, however, we tried to use an approach similar to the one they described, and used programs whose specifications could be easily understood. We also may not have a ground truth CRN, but assume the ones we use to start are working properly; we did confirm that there were no faults detected in the stable CRNs.

Third, we removed mutants that ran out of memory. We removed these from all of our analysis meaning there could be some bias against test suites which generate more data. It is possible that some of our errors were faults which could be detected by some test suites and not others.

Last, we wrote our own programs to generate invariants, run the tests, analyze the results and merge our data. There could be faults in these programs, but we have done some validation on these programs and have manually checked data across/between paper authors. We also showed invariants generated by our program to external users/developers of bioinformatics tools and obtained their feedback as a sanity check. We provide our data artifacts for the community to inspect and re-validate.

5 RESULTS

We now present the results of our three research questions.

Table 3: Fault Detection for invariant flows vs. specification based tests. Columns show the number of concrete test cases and the percent of mutants detected by at least one test. The last two columns indicate mutants detected by the invariant only (Inv.) or specification only (Spec.) tests. A star indicates a small positive value. Org is the non-faulty program.

	Flows			Specification				
CRN	**Org %**	**# Tests**	**% Detected**	**Org %**	**# Tests**	**% Detected**	**# Inv. Only**	**# Spec. Only**
Min	0	92	95.7	0	430	43.5	12	0
S	0	46	87.5	0	354	90.6	2	0
AM	0	93	62.1	0*	279	100.0	0	3
Mod	0	0	—	0	74	87.5	1	0
Max	0	92	87.5	0	160	95.8	1	1
AL1	0	56	59.1	0	262	86.4	2	2
H1	0	30	85.3	0	70	97.1	1	1
H2	0	30	76.0	0	22	92.0	2	3
MWT	0	15	72.7	0*	27	100.0	0	4

5.1 RQ1: Effectiveness of Flow Invariants

To answer this question, we examine Table 3. The table comprises test data using the flow invariants. It compares this with the specification derived tests. The last two columns show the number of mutants detected by invariant tests only, followed by those detected by specification tests only. Within the Flows and Specifications test columns, we show the number of concrete tests performed, the percent of mutants detected, and the percent of mutants that are detected flakily. The first column (Org) is the percent of test failures in the non-faulty program. In all but two subjects (AM and MWT) all concrete test cases pass on all iterations for both techniques. In AM and MWT we saw a very small percentage (less than .5) tests failing flakily on the specification based tests which is expected behavior. A test is flaky if at least one test differs in its detection on one of the 100 runs. The specification based tests find a higher percentage of faults on all subjects but the Min subject. For instance, the invariant tests for H1 detect 85.3% of the mutants while the specification tests detect 97.1%. We see similar data for other subjects such as AM and MWT. However, the invariant tests still find a high percentage of faults (except for Mod). They also have fewer concrete test cases (46 vs. 354 in subtraction and 92 vs. 160 in Max). We examine the implications for tests execution time in RQ2. Mod is interesting since it has no Flow invariants. In order to test this program, we need either inequalities or reaction counting.

If we examine the last two columns we see that in some subjects such as AM and MWT the invariant tests do not find faults that the specification based tests find. We examined some of those mutants (see discussion) and determined that these are mutations which cause all reactions to stop firing. The invariant tests will always hold, but the functional specifications will fail.

Summary of RQ1. We conclude that the flow invariants are effective at fault detection. They also pass on the original programs providing evidence that they accurately describe program behavior. However, they do not perform as well as the specification based test cases in most cases.

5.2 RQ2: Impact of Other Invariants and Reaction Counting

We now examine how adding inequality (with and without reaction counting) and irreducible tests impacts the effectiveness and efficiency of fault detection. Table 4 shows results of the various invariants and the different techniques (with and without reaction counting). The columns for each of these techniques show the percent of faults detected deterministically and flakily, and the number of invariants of that type. For the Subtraction CRN the detection rate improves from 87.5% to 90.6% when we add inequalities, however inequalities on their own are not that effective. If we add reaction counting, the inequalities alone jumps to 75%, and can be combined with the irreducibles which then detects 96.9% of faults. This improves over the specification tests. The trend is consistent across most subjects. The Mod CRN which generated no flows at all, improves to 91.7% fault detection with reaction counting, and is better than the specification tests. For the last four subjects (to be examined further in RQ3), we see both H4 and ECG improve with the additional techniques. In general we find that the invariant tests exhibit slightly lower percentage of flakiness. This is not entirely unexpected since they lack timing information.

We examine efficiency (Table 5) using testing runtime which generally increases for the larger test suite as we add new invariants, but not significantly. However, we do see a jump in time between techniques when we use reaction counting on the larger subjects. Running the `chem_flows` tool takes less than a second for all subjects.

Table 4: Fault Detection by Invariant Type (Flows, Inequality, Irreducible) and Technique (With/Without Reaction Counting). Percents shown for Detect (Det) and Flaky only (flky) detection. Number of invariants (No. Inv). A dash means the type of test is not present for the subject.

	Without Reaction Counting									With Reaction Counting												
	Flows			Ineq			Flows+Ineq			Ineq			Irred			Ineq+Irred			Flow+Ineq+Irred			Specification	
CRN	% Det	% Fkly	No. Inv	% Det	%. Flky	No. Inv	% Det	% Flky	No. Inv	% Det	% Flky	No. Inv	% Det	%. Flky	No. Inv	% Det	% Flky	No. Inv	% Det	% Flky	No. Inv	% Det	% Flky
Min	95.7	0.0	2	8.7	0.0	1	95.7	0.0	3	30.4	0.0	1	—	—	0	30.4	0.0	1	95.7	0.0	3	43.5	0.0
S	87.5	0.0	1	25.0	6.3	2	90.6	3.1	3	75.0	0.0	2	—	—	0	75.0	0.0	2	96.9	0.0	3	90.6	3.1
AM	62.1	3.4	1	—	—	0	62.1	3.4	1	—	—	0	82.1	7.1	2	82.1	7.1	2	96.6	6.9	3	100.0	10.3
Mod	—	—	0	20.8	4.2	2	20.8	4.2	2	87.5	4.2	2	83.3	4.2	1	91.7	4.2	3	91.7	4.2	3	87.5	62.5
Max	87.5	0.0	2	45.8	8.3	3	87.5	0.0	5	83.3	0.0	3	—	—	0	83.3	0.0	3	95.8	0.0	5	95.8	13.0
AL1	59.1	4.5	2	63.6	4.5	5	86.4	4.5	7	77.3	9.1	5	—	—	0	77.3	9.1	5	86.4	4.5	7	86.4	31.8
H1	85.3	0.0	3	23.5	11.8	6	85.3	0.0	9	88.2	0.0	6	38.2	0.0	2	91.2	0.0	8	100.0	0.0	11	97.1	8.6
H2	76.0	0.0	3	28.0	4.0	6	76.0	0.0	9	84.0	0.0	6	44.0	0.0	3	84.0	0.0	9	88.0	0.0	12	92.0	0.0
MWT	72.7	0.0	5	—	—	0	72.7	0.0	5	—	—	0	71.4	0.0	11	71.4	0.0	11	86.4	0.0	16	100.0	9.1
PP	45.8	4.2	5	—	—	0	45.8	4.2	5	—	—	0	91.3	8.7	22	91.3	8.7	22	95.8	8.7	27	—	—
ECG	24.0	0.0	11	0.0	0.0	12	24.0	0.0	23	20.0	0.0	12	—	—	0	20.0	0.0	12	28.0	0.0	23	—	—
H4	64.0	44.0	11	28.0	24.0	16	64.0	44.0	27	52.0	48.0	16	60.0	52.0	23	60.0	52.0	39	64.0	44.0	50	—	—
EC	12.0	0.0	70	4.0	0.0	368	12.0	0.0	438	8.0	0.0	368	8.0	0.0	58	12.0	0.0	426	20.0	0.0	496	—	—

Table 5: Time Required to Run Tests. The table shows the total runtime for all tests across all mutants (Time) in minutes(m) or hours(h).

	Without Reac. Count	With Reac. Count	
CRN	Flows	Flows + Ineq	Flows + Ineq + Irred	Specification
Min	13m	25m	31m	34m
S	32m	1.0h	59m	56
AM	34m	34m	1.3h	1.2h
Mod	—	33m	1.1h	18m
Max	15m	29m	31m	41m
AL1	21m	48m	51m	56m
H1	57m	1.4h	1.8h	47m
H2	12m	28m	46m	50m
MWT	51m	51m	1.7h	1.1h
PP	30m	41m	58m	—
ECG	8m	16m	22m	—
H4	37m	1.1h	2.5h	—
EC	5.0h	18.2h	54.3h	—

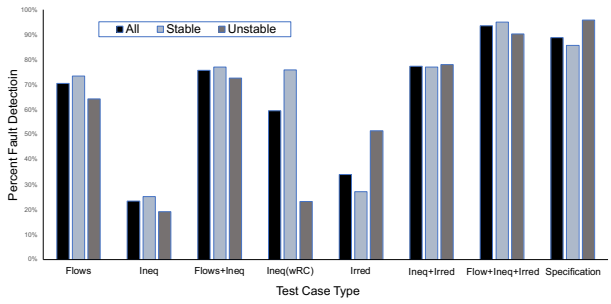

Figure 2: Fault Detection by Type of Test Case. *All* contains all subjects with specification based tests. *Stable* includes subjects Min, S, Mod, Max, H1, H2. *Unstable* includes AM, AL1, MWT.

We also examined the overall fault detection by invariant type for the programs with specification tests (from top half of Table 4). Figure 2 shows this data for all subjects (first bar in each group), followed by stable program (second bar) and unstable programs. Stable programs are guaranteed to return the same result every time. Unstable programs have a probabilistic element. Flow invariants alone (leftmost set of bars) perform pretty well, but detect fewer faults than specification tests (right most group). We do see a slight increase when adding inequalities (third group of bars). When we add reaction counting and all invariants (Flows+Ineq+Irred) next to last group, we see higher fault detection for the *All* and *stable* CRNs over the specification tests. However, the invariants perform worse on the unstable CRNs. This is likely due to the fact that unstable CRNs contain a probabilistic component and extracted invariants are weak since any invariant must account for the probabilistic nature. We explore this issue as future work.

To analyze coverage of the CRNs we examine Figure 3. This shows coverage of individual invariants for the starred subjects in Table 2. For each we show the the normalized reaction coverage (out of the total reactions for that CRN) annotated with an R (e.g. FlowR) and the species coverage annotated with an S (e.g. FlowS). A reaction is covered if a species from an invariant is in the reaction. We see a range of coverage by invariant type suggesting that there may be an opportunity for test selection or prioritization.

We examined one subject further to look for a correlation between coverage fault detection. Table 6 shows 11 invariant tests generated by ChemFlow for the Hailstone 1 subject. We see some correlation between increased species and reaction coverage but leave a full evaluation as future work.

> **Summary of RQ2.** We conclude that adding inequalities slightly improves fault detection. The biggest impact is adding reaction counting. We do see a tradeoff with respect to fault detection and runtime in the larger subjects.

5.3 RQ3: Lack of Specifications

Our last RQ investigates the situation when we do not have specifications, and evaluates the potential to use invariants for model quality. We first examine the data from Tables 4 and 5 focusing on the last four rows. We were able to find faults in all four of these

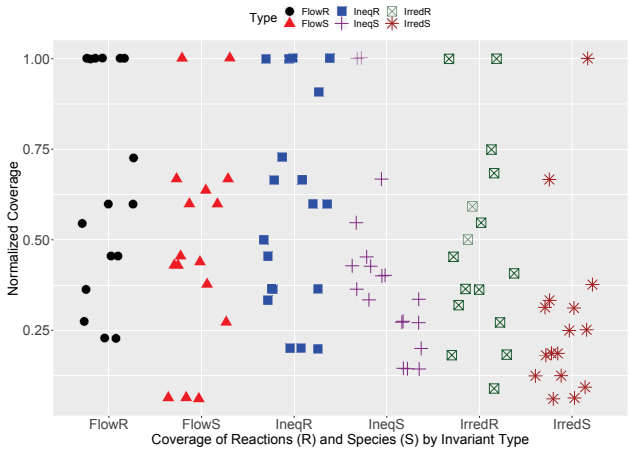

Figure 3: Invariant Coverage by Type

Table 6: Fault Detection by Invariant for Hailstone 1. Type of invariant and technique shown. Ineq-RC indicates inequalities using reaction counting. For each we show the normalized Species Coverage (SpCv) and Reaction coverage (ReCv).

Test ID	Inv Type	% Faults Detected	% SpCv	% ReCv
1	Flow	35	45	45
2	Flow	59	64	73
3	Flow	38	27	36
4	Ineq	3	36	45
5	Ineq	3	45	73
6	Ineq	3	55	91
7	Ineq	12	27	36
8	Ineq	3	27	36
9	Ineq	3	27	36
4	Ineq-RC	35	36	45
5	Ineq-RC	41	45	73
6	Ineq-RC	53	55	91
7	Ineq-RC	29	27	36
8	Ineq-RC	18	27	36
9	Ineq-RC	18	27	36
10	Irred	29	18	36
11	Irred	18	9	18

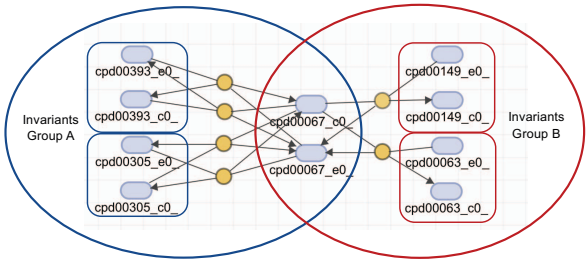

Figure 4: Invariants in *E. coli* represent Islands of Species

subjects. We see similar trends that we saw in RQ2 with respect to fault detection and time.

In addition to running the tests, we were curious if the invariants could provide useful information to bioinfomaticists (external to our organization) who write and use metabolic modeling applications. We met and presented our results using a subset of invariants which we believed might be unexpected based on our understanding of biology. We used the real chemical compound names that these reactions represent. The tool authors suggested that these were interesting and might represent places where our model was missing information or flows and that they would be interested in seeing a fuller set of these on an alternative model (one which they said was more highly curated), suggesting that fewer invariants would mean a better model.

We extracted invariants from a second (higher quality) model of the same variant of *E. coli*. The original model (EC) contains 426 reactions, 497 species, 47 genes and has 70 invariants. We chose the iML1515 model from the UCSD's BiGG database, thought to be the most complete genome scale metabolic model for *E. coli* K-12 MG1655 [27]. It contains 2712 reactions, 1877 metabolites, 1516 genes and has only 32 invariants. When comparing invariants between models, we noticed the refinement of some. In general, the invariants in the iML1515 model are composed of less familiar metabolites suggesting it fixed problems in the EC model. There are also fewer species in the invariants. We showed these to the developers again and they asked us (1) perform a study on hundreds of other models and (2) help incorporate this into one of their systems, which has over 21,000 users, to aid users in the analysis of their models. A real (confirmed) fault in our original *E. coli* subject model is the absence of an exchange flux for fluoride which means it can never leave the cell which is not biologically accurate.

Many of the invariants found show *islands* of species. Figure 3 shows four invariants, denoted by the boxed reactions, from the *E. coli* model. The species on the left (group A) are involved in two reactions whereas the species on the right (group B) are only part of one reaction in the model. The species in group A are vitamins participating in bidirectional transport: each direction is considered a separate reaction. In group B, the species are ions undergoing unidirectional transport (in this case, into the cell). The species in the middle are extracellular protons as well as protons in the cytoplasm necessary for the species to be transported across the cell wall. Since the species in this example are only participating in transport, they could suggest a problem with the model. This is not immediately clear in the network graph, suggesting the invariants are useful.

> **Summary of RQ3.** We conclude that invariants have potential for both regression testing and model quality when no specifications exist.

5.4 Discussion

We discuss a few interesting phenomena we saw during the experiments here.

Specification Tests Find Faults, Invariants Do Not. We found cases where none of the invariants were able to detect a fault, but the specification tests did. An example is the mutant MIN-25. This mutation causes Reaction 1 to add a reactant Z which changes the first reaction ($X + Y \rightarrow Z$) to: $X + Y + Z \rightarrow Z$. Since Z is not an input species, it is initially set to 0 molecules for simulation. However, Z

is a reactant in the mutation, so at least 1 molecule is required for the reaction to fire, thus no reactions can fire. The invariant tests fail to detect this, because the oracle only checks if the values are constant, not what their values are; all invariant tests are tautologies. However, the specifications evaluate functional behavior, and those detect the faults. This was seen in other subjects as well, forming a class of mutations that require specifications.

Invariants Find Faults, Specifications Do Not. An example of this situation is AL1-25, which adds the reaction $F1P \longrightarrow A1 + F1P$. This mutation causes the number of $A1$ molecules in the system to increase. Since at least 1 $A1$ has to be present initially to create the $F1P$ for the reactant, the functional behavior of the CRN is maintained (at least 1 $A1$ is present). This fault is detected by the invariants since they check that the number of molecules in the affected species are constant during simulation. We would need an internal test case in the specification suite to detect this.

Invariants Used for Model Quality. We now look at the results from the biological models. The reduced number of the invariants in the iML1515 model over the EC model suggests improvement, however, some problems remain. An example we identified is the absence of an exchange flux. In the iML1515 model the amount of fluoride remains invariant due to the absence of an exchange flux, so the fluoride can never leave the system. While this invariant was not found in the original model, suggesting the presence of a (new) regression fault, we saw similar invariants for other elements such as Zn2+, Cu2+, Cl-, Ca2+, Co2+, and K+ in the original model.

6 RELATED WORK

Much of the research on CRNs defines novel programs to compute or accomplish some behavioral goal [10, 18, 56]. Another direction of work has been to provide domain specific languages to generate CRNs [31, 45, 54]. In this paper we focus on validation of CRNs. We summarize key related work next.

Verification and Testing CRNs. One state of the art technique for verification of CRNs is to use model checking or automated theorem proving, such as PRISM [36, 55]. PRISM does have a statistical model checker, but it does not support LTL path properties and requires translation of the CRN model to its format. Gerten et al. [20] demonstrated its lack of scalability in particular on faulty CRNs. Vasic et al. [55] use Alloy [25] to model and explore a subset of deterministic CRNs (the other CRN semantic). It is possible that they can extend this to generate tests, however they did not perform this step and the CRNs explored are deterministic, not stochastic CRNs. The most closely related work to this paper is ChemTest [20] and CRNRepair (a program repair technique) [40]. However, both use specification-based tests which we have compared against in this paper. ChemFlow can be applied in either of these environments, hence it is complementary to these approaches.

Invariant Detection. Dynamic invariant detection has been used to extract program invariants both from code [16, 37, 46, 58] and via execution logs [6, 7, 38]. Our invariant detection technique does not require logs or dynamic traces. Instead it uses an algebraic representation.

Petri Net Invariants. There is a breadth of earlier work in the Petri net community on the use of linear invariants to determine structural properties; see [50] for a survey. Generally, these structural properties hold regardless of the initial configuration, and unlike model checking, are determined without examining the reachable states of the Petri net. Instead of P–flows, much work has been done in determining and utilizing P–*semi*flows, which are the P–flows where all coefficients are non-negative. However, while the set of all P–flows can be generated from a basis of size at most equal to the number of species, a generating set of minimal P–semiflows may be exponentially larger. As such, there are different approaches and heuristics to reduce the computational cost of determining the minimal P–semiflows. Our work on P–flows is computationally much simpler, and can extract linear invariants that cannot be expressed as P–semiflows. Other work has shown that Petri nets are effective for modeling metabolic pathways. Gupta et al. explored the use of Petri nets for metabolic pathway validation by determining the presence of inconsistency or deadlock as well as noting that invariant analysis can identify T-invariants (reversible reactions) and P-invariants (conserved compounds) [21].

Many tools utilize Petri nets to analyze their structure [5, 14, 57]. Since CRNs can be translated into a Petri net representation, it is theoretically possible to convert the CRN to a Petri net representation, perform analysis, and then convert the result back to a form corresponding to the CRN. This technique has several disadvantages. (1) As noted above, most Petri net tools build generator sets for P–semiflows, not P-flows, with potentially much higher computational cost and loss of some linear invariants. (2) ChemFlow uses reaction counting, which most Petri net tools do not provide. (3) The ChemFlow framework is designed for developers of CRNs who design, simulate, and validate their systems using the CRN model. Translating the CRN to the Petri net model for testing would force them to use a potentially unfamiliar representation. In addition, it may be difficult to maintain traceability between invariants and the CRN since Petri net tools that translate the analysis back to a CRN model may change names and labels for reactions and species.

7 CONCLUSIONS AND FUTURE WORK

In this paper we presented ChemFlow, an automated invariant inference technique for CRNs. We evaluated ChemFlow on a set of 13 benchmark CRNs including 4 which do not have specifications. We show that the base type of invariant, flows, finds a large number of faults efficiently, but is slightly less effective than specification based test suites. As we add additional types of invariants and reaction counting, the invariants find a similar number of faults as specifications and more deterministically. We also find differences in fault detection by test type when we classify programs as either stable and unstable. Last we see a tradeoff between test case efficiency and effectiveness of the various approaches.

For future work we plan to implement ChemFlow as a tool for others to utilize. We plan to examine the connection between fault detection and CRN coverage in more depth. Finally, we plan to explore model quality more thoroughly by analyzing large set of biological models.

8 ACKNOWLEDGMENTS

We would like to thank C. Henry, J.P. Faria and B. Cottingham for feedback and discussions on the *E. coli* invariants. This work is supported in part by NSF Grants CCF #1909688 and FET #1900716.

REFERENCES

[1] Ebbe S. Andersen, Mingdong Dong, Morten M. Nielsen, Kasper Jahn, Ramesh Subramani, Wael Mamdouh, Monika M. Golas, Bjoern Sander, Holger Stark, Cristiano L. P. Oliveira, Jan Skov Pedersen, Victoria Birkedal, Flemming Besenbacher, Kurt V. Gothelf, and Jørgen Kjems. 2009. Self-assembly of a nanoscale DNA box with a controllable lid. *Nature* 459, 7243 (01 May 2009), 73–76. https://doi.org/10.1038/nature07971

[2] 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*. Association for Computing Machinery, New York, NY, USA, 402–411. https://doi.org/10.1145/1062455.1062530

[3] Adam P. Arkin, Robert W. Cottingham, Christopher S. Henry, Nomi L. Harris, Rick L Stevens, Sergei Maslov, et al. 2018. KBase: The United States Department of Energy Systems Biology Knowledgebase. *Nature Biotechnology* 36, 7 (2018), 566–569. https://doi.org/10.1038/nbt.4163

[4] 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 *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

[5] B. Berthomieu, P.-O. Ribet, and F. Vernadat. 2004. The tool TINA – Construction of abstract state spaces for Petri nets and time Petri nets. *International Journal of Production Research* 42, 14 (2004), 2741–2756. https://doi.org/10.1080/00207540412331312688

[6] Ivan Beschastnikh, Yuriy Brun, Jenny Abrahamson, Michael D. Ernst, and Arvind Krishnamurthy. 2015. Using declarative specification to improve the understanding, extensibility, and comparison of model-inference algorithms. *IEEE Transactions on Software Engineering* 41, 4 (April 2015), 408–428. https://doi.org/10.1109/TSE.2014.2369047

[7] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D. Ernst. 2011. Leveraging Existing Instrumentation to Automatically Infer Invariant-Constrained Models. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 267–277. https://doi.org/10.1145/2025113.2025151

[8] BioBricks Foundation 2021. BioBrick Public DNA Parts. BioBricks Foundation. Last Accessed: August 2021.

[9] Robert Brijder. 2019. Computing with chemical reaction networks: a tutorial. *Natural Computing* 18, 1 (2019), 119–137. https://doi.org/10.1007/s11047-018-9723-9

[10] Luca Cardelli, Marta Kwiatkowska, and Luca Laurenti. 2018. Programming discrete distributions with chemical reaction networks. *Natural Computing* 17, 1 (01 Mar 2018), 131–145. https://doi.org/10.1007/s11047-017-9667-5

[11] Mikaela Cashman, Justin Firestone, Myra B. Cohen, Thammasak Thianniwet, and Wei Niu. 2021. An Empirical Investigation of Organic Software Product Lines. *Empirical Software Engineering* 26, 3 (2021), 44. https://doi.org/10.1007/s10664-021-09940-0

[12] Ho-Lin Chen, Rachel Cummings, David Doty, and David Soloveichik. 2014. Speed Faults in Computation by Chemical Reaction Networks. In *Distributed Computing*, Fabian Kuhn (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 16–30.

[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] G. Chiola, G. Franceschinis, R. Gaeta, and M. Ribaudo. 1995. GreatSPN 1.7: Graphical editor and analyzer for timed and stochastic Petri nets. *Performance Evaluation* 24, 1 (1995), 47–68. https://doi.org/10.1016/0166-5316(95)00008-L Performance Modeling Tools.

[15] Matthew Cook, David Soloveichik, Erik Winfree, and Jehoshua Bruck. 2009. *Programmability of Chemical Reaction Networks*. Springer Berlin Heidelberg, Berlin, Heidelberg, 543–584. https://doi.org/10.1007/978-3-540-88869-7_27

[16] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. 2008. DySy: Dynamic Symbolic Execution for Invariant Inference. In *Proceedings of the 30th International Conference on Software Engineering* (Leipzig, Germany). Association for Computing Machinery, New York, NY, USA, 281–290. https://doi.org/10.1145/1368088.1368127

[17] Shawn M. Douglas, Ido Bachelet, and George M. Church. 2012. A Logic-Gated Nanorobot for Targeted Transport of Molecular Payloads. *Science* 335, 6070 (2012), 831–834. https://doi.org/10.1126/science.1214081

[18] 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 Transactions on Software Engineering Methodology* 28, 2, Article 6 (March 2019), 20 pages. https://doi.org/10.1145/3295740

[19] Martin Feinberg. 2019. *Foundations of chemical reaction network theory*. Springer.

[20] 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 *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 548–560. https://doi.org/10.1145/3324884.3416638

[21] Sakshi Gupta, Sunita Kumawat, and Gajendra Pratap Singh. 2022. Validation and Analysis of Metabolic Pathways Using Petri Nets. In *Soft Computing: Theories and Applications*. Springer, 361–374.

[22] Christopher S Henry, Matthew DeJongh, Aaron A Best, Paul M Frybarger, Ben Linsay, and Rick L Sevens. 2010. High-throughput generation, optimization and analysis of genome-scale metabolic models. *Nature Biotechnology* 28, 9 (August 2010), 977–982.

[23] iGEM Registry 2021. Registry of Standard Biological Parts. iGEM Foundation. Last Accessed: August 2021.

[24] International Society for Nanoscale Science 2021. International Society for Nanoscale Science, Computation and Engineering (ISNSCE). https://isnsce.org/ Accessed on 2021-09-09.

[25] D. Jackson. 2012. *D. Jackson, Software Abstractions, 2nd ed. MIT Press, 2012.* (2nd ed.). MIT Press.

[26] Shuoxing Jiang, Zhilei Ge, Shan Mou, Hao Yan, and Chunhai Fan. 2021. Designer DNA nanostructures for therapeutics. *Chem* 7, 5 (2021), 1156–1179. https://doi.org/10.1016/j.chempr.2020.10.025

[27] Monk JM, Lloyd CJ, Brunk E, and et al. 2017. iML1515, a knowledgebase that computes Escherichia coli traits. *Nat Biotechnol.* 35 (2017), 904–908. https://doi.org/10.1038/nbt.3956

[28] Minoru Kanehisa and Susumu Goto. 2000. KEGG: kyoto encyclopedia of genes and genomes. *Nucleic acids research* 28, 1 (2000), 27–30.

[29] Richard M. Karp and Raymond E. Miller. 1969. Parallel program schemata. *J. Comput. System Sci.* 3, 2 (1969), 147–195. https://doi.org/10.1016/S0022-0000(69)80011-5

[30] KBase 2021. The Department of Energy Systems Biology Knowledgebase. http://kbase.us.

[31] Titus H. Klinge, James I. Lathrop, Sonia Moreno, Hugh D. Potter, Narun K. Raman, and Matthew R. Riley. 2020. ALCH: An Imperative Language for Chemical Reaction Network-Controlled Tile Assembly. In *26th International Conference on DNA Computing and Molecular Programming (DNA 26) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 174)*, Cody Geary and Matthew J. Patitz (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 6:1–6:22. https://doi.org/10.4230/LIPIcs.DNA.2020.6

[32] Marta Kwiatkowska, Gethin Norman, and David Parker. 2010. Advances and challenges of probabilistic model checking. In *2010 48th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. 1691–1698. https://doi.org/10.1109/ALLERTON.2010.5707120

[33] Marta Kwiatkowska and Chris Thachuk. 2014. Probabilistic model checking for biology. In *Software Systems Safety*. IOS Press, 165–189.

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

[35] James I. Lathrop, Titus H. Klinge, and Bryce Valley. 2021. personal communication.

[36] 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 *26th International Conference on DNA Computing and Molecular Programming (DNA 26) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 174)*, Cody Geary and Matthew J. Patitz (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 5:1–5:17. https://doi.org/10.4230/LIPIcs.DNA.2020.5

[37] Kaituo Li, Christoph Reichenbach, Yannis Smaragdakis, and Michal Young. 2013. Second-Order Constraints in Dynamic Invariant Inference. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 103–113. https://doi.org/10.1145/2491411.2491457

[38] David Lo and Shahar Maoz. 2009. Mining Scenario-Based Specifications with Value-Based Invariants. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, USA). Association for Computing Machinery, New York, NY, USA, 755–756. https://doi.org/10.1145/1639950.1639999

[39] Matlab-Simbiology User Forum 2020. MATLAB Forum Question on Testing CRNs. https://www.mathworks.com/matlabcentral/answers/593965-unit-testing-on-simbiology-created-model?s_tid=srchtitle_simbiology_53.

[40] Ibrahim Mesecan, Michael C. Gerten, James I. Lathrop, Myra B. Cohen, and Tomas Haddad Caldas. 2021. CRNRepair: Automated Program Repair of Chemical Reaction Networks. In *2021 IEEE/ACM International Workshop on Genetic Improvement (GI)*. 23–30. https://doi.org/10.1109/GI52543.2021.00014

[41] Tadao Murata. 1989. Petri Nets: Properties, Analysis and Applications. *Proc. IEEE* 77, 4 (April 1989), 541–580.

[42] 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

[43] NCBI 2021. National Center for Biotechnology Information (NCBI), National Library of Medicine (US), National Center for Biotechnology Information. https://www.ncbi.nlm.nih.gov/.

[44] T. J. Ostrand and M. J. Balcer. 1988. The Category-partition Method for Specifying and Generating Functional Tests. *Commun. ACM* 31, 6 (June 1988), 676–686.

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

[46] Jeff H. Perkins and Michael D. Ernst. 2004. Efficient Incremental Algorithms for Dynamic Detection of Likely Invariants. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering* (Newport Beach, CA, USA) *(SIGSOFT '04/FSE-12)*. Association for Computing Machinery, New York, NY, USA, 23–32. https://doi.org/10.1145/1029894.1029901

[47] C. Adam Petri and W. Reisig. 2008. Petri net. *Scholarpedia* 3, 4 (2008), 6477. https://doi.org/10.4249/scholarpedia.6477 revision #91647.

[48] William Poole, Ayush Pandey, Andrey Shur, Zoltan A. Tuza, and Richard M. Murray. 2020. BioCRNpyler: Compiling Chemical Reaction Networks from Biomolecular Parts in Diverse Contexts. *bioRxiv* (2020). https://doi.org/10.1101/2020.08.02.233478

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

[50] Manuel Silva, Enrique Terue, and José Manuel Colom. 1998. Linear algebraic and linear programming techniques for the analysis of place/transition net systems. In *Lectures on Petri Nets I: Basic Models: Advances in Petri Nets*, Wolfgang Reisig and Grzegorz Rozenberg (Eds.). Springer Berlin Heidelberg, 309–373. https://doi.org/10.1007/3-540-65306-6_19

[51] 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.

[52] The Mathworks, Inc. 2021. *MATLAB version 9.10.0.1613233 (R2021a)*. The Mathworks, Inc., Natick, Massachusetts.

[53] Marko Vasic, Cameron Chalk, Sarfraz Khurshid, and David Soloveichik. 2020. Deep Molecular Programming: A Natural Implementation of Binary-Weight ReLU Neural Networks. In *Proceedings of the 37th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 119)*, Hal Daumé III and Aarti Singh (Eds.). PMLR, 9701–9711. https://proceedings.mlr.press/v119/vasic20a.html

[54] 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.

[55] Marko Vasic, David Soloveichik, and Sarfraz Khurshid. 2020. CRNs Exposed: A Method for the Systematic Exploration of Chemical Reaction Networks. In *26th International Conference on DNA Computing and Molecular Programming (DNA 26) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 174)*, Cody Geary and Matthew J. Patitz (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 4:1–4:25. https://doi.org/10.4230/LIPIcs.DNA.2020.4

[56] Erik Winfree. 2019. Chemical Reaction Networks and Stochastic Local Search. In *DNA Computing and Molecular Programming*, Chris Thachuk and Yan Liu (Eds.). Springer International Publishing, Cham, 1–20.

[57] Karsten Wolf. 2018. Petri Net Model Checking with LoLA 2. In *Application and Theory of Petri Nets and Concurrency*, Victor Khomenko and Olivier H. Roux (Eds.). Springer International Publishing, Cham, 351–362.

[58] Lingming Zhang, Guowei Yang, Neha Rungta, Suzette Person, and Sarfraz Khurshid. 2014. Feedback-Driven Dynamic Invariant Discovery. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 362–372. https://doi.org/10.1145/2610384.2610389