A Framework for Testing Chemical Reaction Networks

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

ABSTRACT

The use of non-traditional computing devices is growing rapidly. One paradigm of interest is chemical reaction networks (CRNs) which can model and use chemical interactions for computation. These CRNs are used to develop programs at the nanoscale for applications such as intelligent drug delivery. In practice, these programs are developed in simulation environments, and then compiled into physical systems. A challenge when designing CRNs for computation is the lack of techniques to verify and validate correctness. In this work, we adapt software testing and repair techniques for use in this domain. In initial work, we designed a testing framework to handle the challenges presented by CRN programs; this includes distributed computation and stochastic behavior. We extended this framework to implement automated program repair of CRN models and automated test generation via program invariants. For future work, we will develop a notion of fault localization for these programs, develop a theory of mutation generation, and address issues regarding flakiness present in this computing paradigm.

CCS CONCEPTS

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

KEYWORDS

CRN, test generation, invariants, program repair

ACM Reference Format:
Michael C. Gerten. 2022. A Framework for Testing Chemical Reaction Networks. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3551349.3559562

1 INTRODUCTION

There is a growing utilization of non-traditional computing paradigms. Many of these paradigms can be specified with high-level programming languages[4, 7, 19, 21, 27]. One paradigm is a Chemical Reaction Network (CRN). CRNs are an abstraction of a traditional model of physical chemistry, which can be compiled into DNA as a cyber-physical program [3, 9, 24]. Additionally, it has been shown that CRNs are Turing universal [22]. As a result, CRNs are used as a nanoscale programming language, capable of computing both computational primitives such as addition, and more complex algorithms such as watchdog timers, state logging and finite automata [5, 6, 10, 11, 15]. Since biological organism's behavior can be expressed with CRNs, these programs are of interest as well to biochemists, who use them to describe physical implementations of both natural and synthetic programs. Recently, CRNs have been formulated to represent neural networks and shown effectiveness for common classification tasks [20, 26]. When examining novel uses of CRNs, as many as half the publications in DNA 2021 [23] and many articles when searching the ACM digital library reference CRNs, with applications such as intelligent drug delivery and nano-structures [2, 8]. Given the many applications of these proposed technologies, it is important to validate and verify program behavior. These programs exhibit challenges to validation.

- There is currently a lack of testing approaches that can handle CRN behavior.
- Evaluating CRNs requires a simulation environment.
- Programs are often stochastic in nature, making the time of computation random.
- CRNs are distributed in their computation, making the order of computation random.
- A subset of CRNs themselves are probabilistic in their output, requiring a different type of oracle to test for correctness.
- There exists inherit flakiness that is present in both the CRN programs and the test suites.

With these challenges, it is important to develop techniques to address the testing of CRNs. In this work, we propose a framework which handles the testing of CRNs, as well as providing techniques to automate test generation and program repair.

2 BACKGROUND

The evaluation of non-traditional computing paradigms is often lacking compared to traditional computation. CRNs are one of many emerging paradigms, first used to analyze chemical reactions [3]. A CRN is defined as a set of species S and reactions R. A reaction is composed of reactants on the left side and products on the right side which are sets of species, and a rate constant that determines the speed of the reaction. A common semantic for CRNs is the law of mass-action, which we focus on the stochastic semantics. The state of the model is determined by a Markov process and molecules of each species have natural number counts. These models are evaluated via simulation, in environments such as Matlab's Simbiology package[25]. Consider the CRN given by

$$X1 \xrightarrow{1} Y$$

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

In this example, X2 is subtracted from X1 (X1 − X2) and the results accumulated in the species Y. The rate constants are 1 in both reactions. In the first reaction, X1 is the reactant, and Y is the product. In

This work is licensed under a Creative Commons Attribution International 4.0 License.

ASE '22, October 10–14, 2022, Rochester, MI, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9475-8/22/10.
https://doi.org/10.1145/3551349.3559562

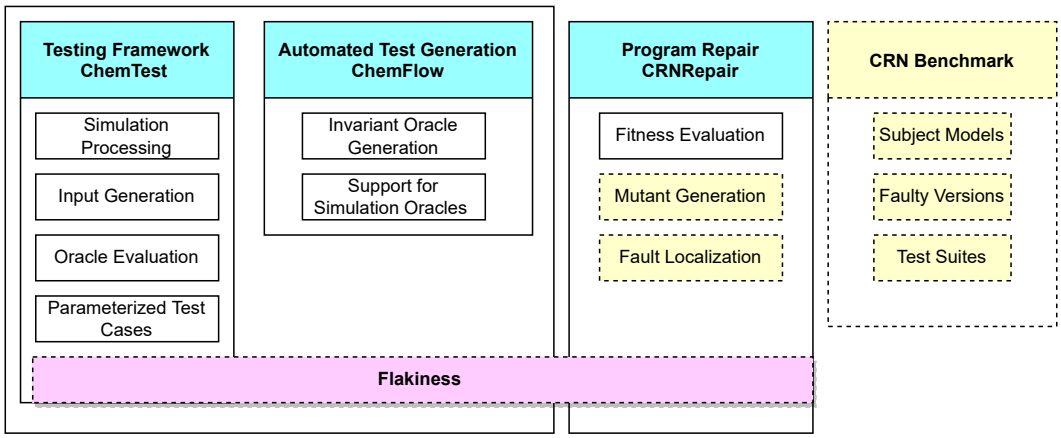

Figure 1: The proposal includes a testing framework of CRNs. We propose extending this framework to include automated test case generation, and mutant generation. These can be used together to implement automated program repair. We further want to construct a benchmark of CRN programs and related artifacts. The yellow shaded work is yet to be completed, the pink is partially completed, and white is completed

the second reaction, null specifies there are no products produced. This program is stable, which means it will have a deterministic output if the simulation is given enough time to finish the evaluation. Some CRNs are not stable and always have a non-deterministic output.

As we implement CRNs there is a need to verify and validate behavior. The current state of the art for evaluating CRNs consists of model checking and theorem proving [9, 10]. These approaches demonstrate correctness but have limitations. Model checking can be unable to scale to models above small molecule counts[16]. We demonstrated this in a preliminary study that some faulty subjects caused the Prism model checker to fail to build the model [13]. Theorem proving also has limitations due to domain knowledge which is needed to construct a proof covering a program's behavior. This can be challenging when program behavior is not fully specified. With these limitations of current approaches there is a need to develop a testing-based approach to validate and verify. We know of no testing framework that can handle the challenges of CRNs.

Another area of interest for CRNs is program repair. Just as the CRN themselves are challenging to test, constructing correct CRNs is difficult. One way to help developers is to create an automated program repair (APR) method, making the job of identifying and fixing errors easier. There are many APR methods [12] commonly using search or evolutionary algorithms. APR has primarily been implemented with traditional programming languages, however recent work has proposed its use with Alloy Models [28]. This suggests an opportunity to apply APR to a new domain to help improve program development.

An important aspect of testing CRNs is handling flakiness. Test flakiness in traditional software is defined as a test that can be observed to both pass and fail without changes to the code [18]. This is true of both test cases with CRNs as well as the program variants themselves. Formally, given a test T in a constant test environment E and a set of input species I and output species O. If O differs on at least one of N runs, T is flaky. In order to reason about the programs behavior, we need to address this flakiness.

3 PROPOSAL

In this work, we propose a framework which enables the testing and repair of CRN programs. Figure 1 shows an overview of my proposed research. The first step is a framework itself, which encompasses simulation processing, input generation, oracle evaluation and test case generation. We will extend this framework to include automated test case generation and additional oracle support. Third, we will develop program repair, adding mutant generation and fault localization to the proposed CRNRepair framework. Fourth, we propose to create a benchmark of CRN programs, test cases, and faults. Finally, we address flakiness in these programs along with possible solutions. We now describe each part of the framework in detail.

3.1 Testing Framework

This framework involves addressing the challenges of the stochastic CRNs. We must be able to create test cases from the specifications of the model. This requires a modified first-order logic which can express both the functional properties as well as the temporal properties that a CRN can be required to satisfy. These can be instantiated with concrete input values and generated based on constraints of the specifications. This forms oracles to be evaluated via simulation. Due to the distributed and concurrent nature of these programs, we have to handle the evaluation time and how the results are affected. Additionally, due to the inherit flakiness of the models, we must evaluate each test a number of times in order to observe model behavior. The choice of these parameters can be set to default values, but may require optimization based on the subject model.

3.2 Automated Test Case Generation

We can then automate portions of this framework. Utilizing invariant techniques used for Petri nets, we can automatically generate

oracles from the program model. This is accomplished by generating the linear invariants and creating oracles from them. Some of the created oracles require simulation information in addition to the output, requiring changes to the underlying framework to accommodate these tests. This enables automated test case generation for regression testing.

3.3 Program Repair

Further, we can use the proposed testing framework to implement program repair on CRNs. A notion of model fitness for CRNs must be created to be able to conduct the search-based repair. We further need to create an abstract syntax tree to work within the genetic improvement framework. To improve the repair process, we propose the development of fault localization. To aid in the search process, new operators will be defined. This will lead to the development of theory of mutant generation for CRN programs. This further formalizes mutation testing of CRN programs. The notion of coverage for CRNs will be defined, as to provide a metric for fault localization and test coverage.

3.4 Benchmark Programs

While there is a large community of researchers building CRNs, we lack a common set of programs and their faulty versions to benchmark testing approaches. This research proposes the creation of a benchmark of CRN programs and related artifacts. This will enable further research without each group needing to recreate a reasonable benchmark set. As CRNs increase their usage, the existence of a general benchmark set will help advancements in the verification and validation. This benchmark will include programs gathered from a literature search, a set of generated faults, and the test suites needed to evaluate the models.

3.5 Flakiness

Finally, we propose an investigation into the role flaky behavior plays in the testing and repair of CRNs. In our initial results, flakiness is a common attribute to be addressed, which we have devised methods in the testing framework. As we further explore the causes of flaky tests and models, we hope to gain techniques that can be used to further improve the testing and repair frameworks. And extend these insights to developers in other programming environments.

4 CONTRIBUTION

In this work the expected contributions are presented below:

- We propose an end-to-end testing framework, ChemTest, for stochastic CRNs, taking models and a set of specifications as input, and evaluating correctness.
- We propose a program repair framework to handle CRNs, implementing changes required to handle this new domain. We evaluate the effect of parameters on the repair process.
- We extend ChemTest by implementing an automated test generation approach, utilizing invariants extracted from the CRN model. This evaluation is in comparison to the manual tests generated in the original framework.
- We complete case studies to evaluate each contribution for its effectiveness and impact on validating CRNs
- We develop a method of fault localization to improve program repair and bug detection in CRN models.
- We compile a benchmark dataset of CRN programs for access to the community.
- We investigate the properties of flakiness in CRN models and see what lessons can be applied to improve testing and repair of CRNs.

Table 1: ChemTest, number of Deterministic, Flaky, and Mixed (having both Flaky and Deterministic) test cases and mutants by Subject. Tests are listed for both Abstract and Concrete, in parentheses [13].

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

5 RESULTS

5.1 Testing Framework: ChemTest

We built an initial version of a software testing framework, ChemTest, which can be integrated into the MATLAB Simbiology framework [13]. It uses a LTL-like logic to represent the specifications and oracles, which in turn define abstract tests. We then use the test specification language to instantiate concrete values for the abstract tests based on constraints in the specifications. Each test then was simulated 100 times and evaluated at simulation time 100 to account for stochastic behavior and time dependency of the output. Our initial results with the ChemTest framework show that effective test suites can be developed for CRNs. Utilizing several types of tests, functional, metamorphic, and hyper, we were able to achieve a high level of fault detection in a set of generated mutant models, finding 80.4% of generated faults using metamorphic tests and 66.5% using functional tests.

Further we explored the effect of flakiness in the subjects, including both flakiness from the stochastic program behavior and the simulation environment. This includes evaluating the effect of time of evaluation on test correctness [13]. When we look at Table 1, we see that the majority of subjects exhibit mixed behavior. This means that some tests are deterministic and some are flaky. Looking closer, we can see that one hailstone mutant was only detected via flaky tests, demonstrating that those tests are required to achieve fault detection on some errors. Further investigation into model behavior under test is needed to understand how to best test CRNs.

5.2 Program Repair: CRNRepair

Fixing faults in CRNs is non-trivial. Due to the stochastic nature of CRNs, bugs in the program can be hard to identify. An example fault from our study subject Hailstone that had 11 reactions can be

as simple as removing a product species A from

$$M \longrightarrow A + A + A + A + A + A + B + B + B.$$

This yields incorrect behavior of the CRN, but only on a small set of inputs, and is visually difficult to see in the full program text. Therefore, in our initial work [13], we modified the PyGGI repair tool [1] to evaluate the ability to repair mutant CRN programs. We implemented a custom abstract syntax tree to parse CRNs, added support for the XML format of CRNs, and added a base set of operators. With these improvements, we could repair faulty programs, 90% of faults for one subject, and 50% for another subject [17]. An interesting attribute of the repairs we found is that they had 3 types of behavior. In the repairs found, the model was either returned to the original correct model behavior, or modified in such a way to speed up or to slow down the computation. While the successful repairs had different simulation behavior, all were functionally correct.

This result has led us to propose developing a more advanced search method which can account for non-functional properties such as evaluation time or number of reactions in the candidate repair. We also evaluated which parameters helped the repair process and found a middle ground of 40 epochs vs 50 iterations to be most effective for the programs evaluated. However, we did not develop fault localization or mutation operators to help guide the program repair, which is work to be completed in my research.

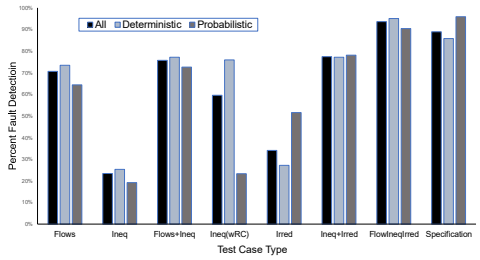

Figure 2: ChemFlow, fault detection by test case type, All contains all subjects. Stable includes Min, S, Mod, Max, H1, H2. Unstable includes AM, AL1, MWT citeChemFlow.

5.3 Automated Test Generation: ChemFlow

Further, CRN programs may not have the specifications needed to create oracles. Biological models built from cumulative experimentation represent program behavior, but lack a known specification from which to build tests from. But even simple functional programs lead to issues when the specifications are unclear, such as a X mod Y function. When developing this function's specifications, we incorrectly assumed that its output would always be a Boolean value, leading to over fitting the test suite and creating false positive failures when it returned integer values. Third, the specifications we used in ChemTest require the developer to use a first-order logic like language such as LTL which may be hard to write.

To address these issues, we have utilized program invariants extracted from the CRN models [14]. We utilize a Petri net formulation of the CRNs to extract the invariants in a tool called ChemFlow. These invariants represent specifications (including oracles) that hold for the entire program. We have defined three types of invariants, in order to create automated program oracles to evaluate behavior. This helps address a shortcoming of the original framework by reducing the need for manually generated specifications. This resulted in a similar level of fault detection when evaluated against a set of mutant programs as the manual specifications from the original ChemTest work[14]. When we look at Figure 2, we can see as we add additional invariant types together, such as Flows and Irreducibles, we gain higher fault detection than either set alone. This shows that the type of test used does affect fault detection and the proposed work to define test coverage is needed to evaluate the strength of CRN test suites.

6 PLAN OF WORK

As we move forward, we plan on completing the following aspects of this work.

- We plan on extending the program repair framework. This will add fault localization, new mutation operators, and search algorithms which will improve the ability to repair more complex CRN programs and faults. We will evaluate these improvements against our current benchmark set used in ChemFlow and previous work CRNRepair [14, 17].
- We plan on creating a benchmark of CRN programs. This benchmark will consist of CRN program model, specifications, testing artifacts, and faults. This will enable the community to further research into validation of CRN programs without needing to recreate a representative set of programs. To expand our initial benchmark used in the ChemFlow work [14], we will conduct a broad literature review to gather subjects.
- We will explore the flaky behavior of CRN programs and what can be learned from them. This will provide a better understanding of the testing results from our work. It will also provide insights into how to better utilize tests that behave flakily in other computing systems.

ACKNOWLEDGMENTS

This work is supported in part by National Science Foundation Grants CCF-1909688 and FET-1900716.

REFERENCES

[1] G. An, A. Blot, J. Petke, and S. Yoo. 2019. PyGGI 2.0: Language independent genetic improvement framework. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1100–1104.

[2] 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 (May 2009), 73–76. https://doi.org/10.1038/nature07971

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

[4] 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/

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

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

[7] 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

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

[9] 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 *14. Association for Computing Machinery, New York, NY, USA.* ASE, 767–778. https://doi.org/10.1145/2642937.2643007

[10] 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 (2019), 6–20. https://doi.org/10.1145/3295740

[11] 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, 2017. Association for Computing Machinery, , NY, USA, 23:1–23:6.* 27–29.

[12] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2019. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* 45, 1 (2019), 34–67. https://doi.org/10.1109/TSE.2017.2755013

[13] 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* (Virtual Event, Australia) *(ASE '20).* Association for Computing Machinery, New York, NY, USA, 548–560. https://doi.org/10.1145/3324884.3416638

[14] 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 *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE).* 1193–1205. https://doi.org/10.1145/3510003.3510176

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

[16] James I. Lathrop, Jack H. Lutz, Robyn R. Lutz, Hugh D. Potter, and Matthew R. Riley. 2021. Population-induced phase transitions and the verification of chemical reaction networks. (2021). https://doi.org/10.1007/s11047-021-09877-9

[17] 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

[18] Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. 2021. A Survey of Flaky Tests. *ACM Trans. Softw. Eng. Methodol.* 31, 1, Article 17 (oct 2021), 74 pages. https://doi.org/10.1145/3476105

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

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

[21] 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::82, 75–9.

[22] Nicholas Schiefer and Erik Winfree. 2015. Universal Computation and Optimal Construction in the Chemical Reaction Network-Controlled Tile Assembly Model. In *DNA Computing and Molecular Programming*, Andrew Phillips and Peng Yin (Eds.). Springer International Publishing, Cham, 34–54.

[23] International Society. [n.d.]. *for Nanoscale Science 2021.* International Society for Nanoscale Science, Computation and Engineering (ISNSCE. https://isnsce.org/

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

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

[26] 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, vasic20a.html, 9701–9711. https://proceedings.mlr.press/v119/

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

[28] Kaiyuan Wang, Allison Sullivan, and Sarfraz Khurshid. 2018. Automated Model Repair for Alloy. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE).* 577–588. https://doi.org/10.1145/3238147.3238162