Initial Results on Counting Test Orders for Order-Dependent Flaky Tests Using Alloy

Wenxi Wang[1(✉)], Pu Yi[2], Sarfraz Khurshid[1], and Darko Marinov[3]

[1] The University of Texas at Austin, Austin, USA
wenxiw@utexas.edu
[2] Peking University, Beijing, China
[3] University of Illinois Urbana-Champaign, Champaign, USA

Abstract. Flaky tests can seemingly nondeterministically pass or fail for the same code under test. Flaky tests are detrimental to regression testing because tests that pass before code changes and fail after code changes do not reliably indicate problems in code changes. An important category of flaky tests is order-dependent tests that pass or fail based on the order of tests in the test suite. Prior work has considered the problem of counting test orders that pass or fail, given relationships of tests within a test suite. However, prior work has not addressed the most general case of these relationships. This paper shows how to encode the problem of counting test orders in the Alloy modeling language and how to use propositional model counters to obtain the count for test orders. We illustrate that Alloy makes it easy to handle even the most general case. The results show that this problem produces challenging propositional formulas for the state-of-the-art model counters.

1 Introduction

Flaky tests [15] can seemingly nondeterministically pass or fail for the same code under test. Flaky tests are detrimental to regression testing because tests that pass before code changes and fail after code changes do not reliably indicate problems in code changes. For example, Harman and O'Hearn point out problems of flaky tests at Facebook [5], and several other companies point out similar problems, including Apple [11], Google [4,16,22], Huawei [9], and Microsoft [6,7,13,14].

An important category of flaky tests is order-dependent tests that pass or fail based on the order of tests in the test suite. More specifically, the tests deterministicaly fail in some test orders and deterministically pass in other test orders. Before establishing that the tests depend just on the order, the developers may view them as nondeterministically passing or failing in various runs.

Shi et al. [17] have categorized several roles for order-dependent tests. Each order-dependent test itself can be either a *victim*, which passes when run by itself but fails when run after some other tests in the test suite, or a *brittle*, which fails when run by itself but passes when run after some other test in the test suite. Each victim test fails when run after (not necessarily immediately after)

Published by Springer Nature Switzerland AG 2022
D. Clark et al. (Eds.): ICTSS 2021, LNCS 13045, pp. 123–130, 2022.
https://doi.org/10.1007/978-3-031-04673-5_9

a *polluter* test, unless a *cleaner* test runs between the polluter and the victim. Each brittle test passes when run after (not necessarily immediately after) a *state-setter* test. We focus on victim tests, because the analysis for brittle tests comes out as a special case.

Wei et al. [20] have recently considered the problem of counting the number of test orders for which a victim fails. This problem is important because it allows computing the *flake rate*, i.e., the probability that a test fails if the test order is a uniformly sampled permutation of the test suite. In turn, the flake rate allows developers to determine whether to fix the test or not, and it allows researchers to compare various algorithms for detecting order-dependent tests [20]. Wei et al. [20] have derived analytical formulas for some cases of victims, namely when all polluters have the same set of cleaners, but have not addressed the most general case, namely when two or more polluters have a different set of cleaners.

We show how to encode the problem of counting test orders in the Alloy modeling language [8], and we use propositional model counters [12,18] to count the test orders. Alloy has been used for many software analysis and testing tasks [3,10], and model counters have seen wide applications in various domains [1,2]. The Alloy toolset automatically translates the Alloy models into propositional formulas that are fed into model counters to solve the counting problems. Yang et al. [21] have presented AlloyMC that connects Alloy with model counters. However, no prior work has used Alloy to count test orders.

We illustrate how Alloy makes it easy to handle even the most general case of victims with polluters that may have different cleaners. We show a general *skeleton* model to encode the problem of counting test orders; the skeleton can be instantiated with the specific sets of polluters and cleaners. To evaluate correctness and scalability of our approach, we use 24 propositional formulas as our benchmarks. The benchmarks consider a real scenario from the flaky test dataset published by Wei et al. [20] with two polluters where one has a subset of cleaners of the other. We instantiate our skeleton with an increasing number of cleaners. We choose Alloy to translate this difficult problem into SAT formula, because the Alloy analyzer employs the heavily optimized constraint solver Kodkod [19], which efficiently translates Alloy specifications into simplified SAT formulas. The SAT formulas can be counted using any off-the-shelf model counters. We apply state-of-the-art model counters for both exact counting (ProjMC [12]) and approximate counting (ApproxMC4 [18]).

The results show that the problem of counting test orders provides *challenging* propositional formulas for model counters. In addition, we found that the exact counter generally runs *faster* than the approximate counter for all our non-trivial benchmarks, which is a surprising result because it is unusual that an exact model counter outperforms an approximate model counter [18].

In summary, this paper makes the following contributions:

- **Encoding:** We show how to encode the problem of test orders in Alloy.
- **Evaluation:** We evaluate our encoding on a number of challenging problems. The initial results are promising but point out to scalability issues.
- **Challenges:** We obtain a number of interesting and challenging problems for propositional model counters.

```
1.  open util/ordering[Test]

2.  abstract sig Test {}
3.  one sig Victim extends Test {}
4.  abstract sig Cleaner extends Test {}
5.  abstract sig Polluter extends Test { cleaners: set Cleaner }
6.  fact { Polluter.cleaners = Cleaner }
7.  pred Pollutes[p: Polluter] {
8.      p in prevs[Victim]
9.      and no p.cleaners & prevs[Victim] & nexts[p] }
10. pred Fail[] {
11.     some p: Polluter | Pollutes[p]
12.     and no p': nexts[p] & prevs[Victim] & Polluter | Pollutes[p'] }
13. pred Pass[] {
14.     !Fail[] }
15. pred Pass2[] {
16.     all p: Polluter & prevs[Victim] |
17.         some p.cleaners & prevs[Victim] & nexts[p] }

18. one sig c_1, c_2, c_3 extends Cleaner {}
19. one sig p_1, p_2 extends Polluter {}
20. fact Matrix {
21.     p_1.cleaners = c_1 + c_2
22.     p_2.cleaners = c_1 + c_2 + c_3 }

23. run Fail
24. run Pass
25. run Pass2
```

Fig. 1. An example of modeling flaky test orders in Alloy

2 Modeling Flaky Test Orders Using Alloy

We illustrate our approach for modeling flaky test orders in Alloy using an example. Through the example we also introduce the aspects of the Alloy language required to understand the modeling. Figure 1 shows an example Alloy model which encodes the problem of counting test orders.

We model the order of tests in a test suite using the Alloy library `util/ordering`, which defines a linear order (line 1). The signature (`sig`) `Test` declares a set of atoms that represent tests (line 2). The set of tests is partitioned (using keyword `extends`) into three subsets: a singleton (`one`) set for the victim (line 3), a set of cleaners (line 4), and a set of polluters (line 5). Note that we do not model neutral tests which do not have any impact on the the victim, because their presence has no impact on the flake rate. The *field* `cleaners` in sig `Polluter` introduces a binary relation `cleaners: Polluter x Cleaner` to represent the matrix that relate polluters to cleaners (line 5). A `fact` introduces a constraint that must be satisfied in all models; the stated `fact` uses relational composition ('.') to require that the relational image of `Polluter` under the

relation `cleaners` equals the set of all cleaners, i.e., models have no extraneous cleaners (line 6).

A predicate (`pred`) introduces a parameterized formula that can be *invoked* elsewhere. The predicate `Pollutes` enforces two constraints on its parameter `p` that is a polluter (lines 7–9). One, `p` appears before the victim in the test order; `prevs[i]` (likewise, `nexts[i]`) is a library function that represents the set of atoms in the linear order before (likewise, after) `i`, and `in` is the subset operator. Two, no cleaner for `p` is between `p` and the victim; the quantifier `no` is the negation of existential quantifier `some`, and '`&`' is set intersection. A vertical bar "—" indicates the start of a sequence of constraints. The predicate `Fail` defines the failing test order using the existential quantification: there is some atom in the set of polluters such that it pollutes (lines 10–12). The additional constraint (line 12) requires that `p` be the *last* such test before the victim, which rules out duplicate solutions where the difference is not based on the test order but based on which polluter is a *witness* to failure. (More formally, this constraint ignores from the model the new variable arising from Skolemization.) The predicate `Pass` defines the passing test order as the negation of the constraints for failing test order (lines 13–14). We also evaluate the predicate `Pass2` that defines another encoding for the passing test order, stating more directly that all polluters before the victim have a cleaner between the polluter and the victim; we expect this encoding to enable faster model counting.

The test suites in a general model contain 1 victim, n polluters, and k cleaners. Figure 1 (lines 18–22) shows one example containing 2 polluters and 3 cleaners. The `Matrix` (`fact`) states the cleaners for each polluter. In this example, polluter `p_1` has two cleaners `c_1` and `c_2`, and polluter `p_2` has three cleaners `c_1`, `c_2`, and `c_3`. We can change the number of polluters and cleaners simply by changing the declarations in lines 18–19, and change the relations between polluters and cleaners by changing the `Matrix`. The `run` command defines the constraint-solving problem, which is to solve the predicate `Fail`/`Pass`/`Pass2` subject to all applicable constraints on the sets and relations declared in the Alloy model (lines 23–25). Each model represents one test order, and counting the number of models thus counts the number of test orders.

3 Experimental Evaluation

```
18. one sig c_1, c_2, ..., c_k extends Cleaner {}
19. one sig p_1, p_2 extends Polluter {}
20. fact Matrix {
21.     p_1.cleaners = c_1 + c_2 + ... + c_(k/2)
22.     p_2.cleaners = c_1 + c_2 + ... + c_k }
```

Fig. 2. The template for our benchmark generation

Table 1. Counting results of our benchmarks ('-' denotes not applicable)

Benchmarks	ProjMC		ApproxMC		
	Time	Count	Time	Count	Error (%)
k = 1, Fail	0.01	15	0.00	15	0.00
k = 2, Fail	0.02	56	0.01	56	0.00
k = 3, Fail	0.09	270	0.21	260	3.85
k = 4, Fail	0.87	1800	1.22	1856	3.11
k = 5, Fail	6.79	12096	15.27	13056	7.94
k = 6, Fail	93.21	104832	204.67	110592	5.49
k = 7, Fail	937.31	907200	1040.32	884736	2.54
k = 1, Pass	0.01	9	0.00	9	0.00
k = 2, Pass	0.02	64	0.01	64	0.00
k = 3, Pass	0.13	450	0.45	496	10.22
k = 4, Pass	1.23	3240	5.21	3456	6.67
k = 5, Pass	13.71	28224	72.8	31744	12.47
k = 6, Pass	256.93	258048	547.6	278528	7.94
k = 7, Pass	3197.09	2721600	>5000	–	–
k = 1, Pass2	0.01	9	0.00	9	0.00
k = 2, Pass2	0.02	64	0.01	64	0.00
k = 3, Pass2	0.14	450	0.44	496	10.22
k = 4, Pass2	1.22	3240	4.74	3456	6.67
k = 5, Pass2	12.33	28224	80.05	31744	12.47
k = 6, Pass2	217.80	258048	555.24	278528	7.94
k = 7, Pass2	2905.76	2721600	4700.48	2359296	15.36

3.1 Setup

Model Counters. We study how both exact model counting and approximate model counting perform on the generated propositional formulas. We apply ProjMC [12], which is the state-of-the-art exact model counter, and ApproxMC4 [18], which is the state-of-the-art approximate model counter.

Benchmarks. As our benchmark, we want to generate propositional formulas for `Fail`, `Pass`, and `Pass2` predicates introduced in the above Alloy model (Fig. 1) with various test combinations. To do so, we replace lines 18–22 of our Alloy model with a simple template shown in Fig. 2. The template introduces 2 polluters and k cleaners: the first polluter has half of all cleaners as its cleaners, and the second polluter has all the cleaners as its cleaners. In our experiments, we range k from 1 to 8, generating 8 propositional formulas for each predicate. In total, we generate 24 benchmarks for our evaluation. We choose to use this template because it is the most complicated case that exists in the real-world

flaky test dataset published by Wei et al. [20]. Therefore, we think it represents the most complicated model we can encounter in a real-life setting.

Metrics. The two key metrics we use in our evaluation are the model counts and the actual wall time to compute them. In line with ApproxMC, we report the error rate of the approximate model counting as $max(\frac{approx}{exact}, \frac{exact}{approx}) - 1$, based on multiplicative guarantees. We use timeout of 5000 s, as commonly done in work on model counting [18].

Platform. All the experiments are conducted on a machine with Intel Core i7-8700k CPU (12 logical cores in total) and 32-GB RAM.

3.2 Results

We apply both model counters on all the generated propositional formulas (with k ranging from 1 to 8) for all the three predicates (i.e., `Fail`, `Pass`, and `Pass2`). Our experimental results show that the limit of ProjMC for all three predicates is $k = 7$; the limit of ApproxMC for `Fail` and `Pass2` is $k = 7$ and for `Pass` is $k = 6$. Thus, the propositional formulas generated with our Alloy model are generally difficult, providing a challenging dataset for future work on model counters. For formulas in many other domains, model counters can handle orders of magnitude more models [18]. For formulas in many other domains, model counters can handle many more models with orders of magnitude [18].

The detailed results of the benchmarks encoding all the three predicates with k up to 7 are shown in Table 1. Our results provide a way to sanity check the correctness of our proposed Alloy model: the exact counts of `Pass`/`Pass2` should be the same, and the sum of the exact count of `Fail` and the exact count of `Pass`/`Pass2` should be the total number of permutations of all the tests (i.e., k cleaners, 2 polluters, and 1 victim), i.e., $(k+3)!$. With the exact counts reported by ProjMC, we confirm that our proposed model passes the sanity check. We also manually check that correct models (not just count) are generated for $k = 1$.

Moreover, the results show that ProjMC generally runs *faster* than ApproxMC for all the non-trivial benchmarks (where $k \geq 3$), which is quite a surprising result. It is unusual that an exact model counter outperforms an approximate model counter. Hence, our non-trivial benchmarks pose additional value for model counting community. Our intuitive explanation for this phenomenon is that the problem of counting the test orders for order dependent flaky tests is generally hard and has little space for the optimization, which may not favor the fancy tricks operated by the approximate model counters as ApproxMC. We also did some preliminary experiments using the SAT encoding, this phenomenon still exists. Therefore, we do not think it is the encoding Alloy provides that causes this interesting results. Besides, we can also observe that ApproxMC sometimes over-approximates and sometimes under-approximates, with the error rate ranging from 0.00% to 15.36%; interestingly, ApproxMC approximates with lower error for the `Fail` predicate than for the `Pass`/`Pass2` predicates. Lastly, the results show that the `Pass` predicate is easier for both

counters to solve, compared to the `Pass2` predicate. Therefore, we confirm that different encoding for the same problem can result in different counting efficiency.

4 Conclusions

This paper presents a general way of encoding the problem of counting flaky test orders using the Alloy modeling language. We illustrate how Alloy makes it easy to handle even the most general case of victims with polluters that have different cleaners. We provide a general skeleton Alloy model that can be instantiated with the specific sets of polluters and cleaners. To evaluate our encoding, we generate 24 problems with various test combinations of different sizes. The results show that our Alloy encoding provides interesting and challenging propositional formulas that can be a useful resource to advance development of the state-of-the-art model counters.

Our future work is to evaluate more encodings of passing and failing test orders to try to improve scalability of the approach. We hope that we can push the approach to $k = 10$. Such values would cover many real cases [20] and also provide inspiration to develop analytical formulas for the number of test orders.

Acknowledgment. We thank Wing Lam and Anjiang Wei for discussions on counting test orders. This work was partially supported by NSF grants CCF-1763788. We also acknowledge support for research on flaky tests from Facebook and Google.

References

1. Aydin, A., Bang, L., Bultan, T.: Automata-based model counting for string constraints. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015, Part I. LNCS, vol. 9206, pp. 255–272. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_15
2. Bacchus, F., Dalmao, S., Pitassi, T.: Algorithms and complexity results for # SAT and Bayesian inference. In: FOCS (2003)
3. Büttner, F., Egea, M., Cabot, J., Gogolla, M.: Verification of ATL transformations using transformation models and model finders. In: Aoki, T., Taguchi, K. (eds.) ICFEM 2012. LNCS, vol. 7635, pp. 198–213. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34281-3_16
4. Google: Avoiding flakey tests (2008). http://googletesting.blogspot.com/2008/04/tott-avoiding-flakey-tests.html
5. Harman, M., O'Hearn, P.: From start-ups to scale-ups: opportunities and open problems for static and dynamic program analysis. In: SCAM (2018)
6. Herzig, K., Greiler, M., Czerwonka, J., Murphy, B.: The art of testing less without sacrificing quality. In: ICSE (2015)
7. Herzig, K., Nagappan, N.: Empirically detecting false test alarms using association rules. In: ICSE (2015)
8. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press, Cambridge (2006)
9. Jiang, H., Li, X., Yang, Z., Xuan, J.: What causes my test alarm? Automatic cause analysis for test alarms in system and integration testing. In: ICSE (2017)

10. Kang, E., Jackson, D.: Formal modeling and analysis of a flash filesystem in alloy. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) ABZ 2008. LNCS, vol. 5238, pp. 294–308. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87603-8_23
11. Kowalczyk, E., Nair, K., Gao, Z., Silberstein, L., Long, T., Memon, A.: Modeling and ranking flaky tests at Apple. In: ICSE-SEIP (2020)
12. Lagniez, J.-M., Marquis, P.: A recursive algorithm for projected model counting. In: AAAI, vol. 33, pp. 1536–1543 (2019)
13. Lam, W., Godefroid, P., Nath, S., Santhiar, A., Thummalapenta, S.: Root causing flaky tests in a large-scale industrial setting. In: ISSTA (2019)
14. Lam, W., Muşlu, K., Sajnani, H., Thummalapenta, S.: A study on the lifecycle of flaky tests. In: ICSE (2020)
15. Luo, Q., Hariri, F., Eloussi, L., Marinov, D.: An empirical analysis of flaky tests. In: FSE (2014)
16. Memon, A., Gao, Z., Nguyen, B., Dhanda, S., Siemborski, R., Micco, J.: Taming Google-scale continuous testing. In: ICSE-SEIP, Eric Nickell (2017)
17. Shi, A., Lam, W., Oei, R., Xie, T., Marinov, D.: iFixFlakies: a framework for automatically fixing order-dependent flaky tests. In: FSE (2019)
18. Soos, M., Gocht, S., Meel, K.S.: Tinted, detached, and lazy CNF-XOR SOLVING and its applications to counting and sampling. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020, Part I. LNCS, vol. 12224, pp. 463–484. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53288-8_22
19. Torlak, E., Jackson, D.: Kodkod: a relational model finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71209-1_49
20. Wei, A., Yi, P., Xie, T., Marinov, D., Lam, W.: Probabilistic and systematic coverage of consecutive test-method pairs for detecting order-dependent flaky tests. In: TACAS 2021, Part I. LNCS, vol. 12651, pp. 270–287. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72016-2_15
21. Yang, J., Wang, W., Marinov, D., Khurshid, S.: Alloy meets model counting. In: FSE, AlloyMC (2020)
22. Ziftci, C., Reardon. J.: Who broke the build? Automatically identifying changes that induce test failures in continuous integration at Google scale. In: ICSE (2017)