FLACK: Localizing Faults in Alloy Models

Guolong Zheng[‡], ThanhVu Nguyen[¶], Simón Gutiérrez Brida[*†],
Germán Regis[*], Marcelo Frias[†§], Nazareno Aguirre[*†], Hamid Bagheri[‡]
[*]Department of Computer Science, FCEFQyN, University of Río Cuarto, Argentina
[†]National Council for Scientific and Technical Research (CONICET), Argentina
[‡]Department of Computer Science & Engineering, University of Nebraska-Lincoln, USA
[§]Department of Software Engineering, Buenos Aires Institute of Technology, Argentina
[¶]Department of Computer Science, George Mason University, USA

Abstract—**Fault localization can help developers identify buggy statements or expressions in programs. Existing fault localization techniques are often designed for imperative programs (e.g., C and Java) and rely on tests to compare correct and incorrect execution traces to identify suspicious statements. In this demo paper, we present FLACK, a tool to automatically locate faults for models written in Alloy, a declarative language where the models are not executed but instead converted into a logical formula and solved using a SAT solver. FLACK takes as input an Alloy model that violates some assertions and returns a ranked list of suspicious expressions contributing to the violation. The key idea is to analyze the differences between *counterexamples*, i.e., instances of the model that do not satisfy the assertion and instances that do satisfy the assertion to find suspicious expressions in the input model. An experiment with 157 Alloy models with various bugs shows the efficiency and accuracy of FLACK in localizing the causes of these bugs. FLACK and its evaluation benchmark and results can be downloaded from https://github.com/guolong-zheng/flack. The video demonstration is available at https://youtu.be/FKa2ohqIUms.**

Index Terms—**Alloy, fault localization, specifications, models**

I. Introduction

The Alloy specification language [1] has been used for various software modeling and analysis tasks such as program verification [2], test case generation [3], network security [4], and security analysis of IoT and Android platforms [5], [6]. Similar to developing programs in an imperative language like C or Java, developers can make subtle mistakes when using Alloy in modeling system specifications, especially those that capture complex systems with non-trivial behaviors. However, traditional fault localization techniques for imperative programs do not directly apply to a specification language such as Alloy, in which there are no control flow graphs or program execution traces often used to aid debugging.

To aid developers in debugging Alloy models, in [7] we introduce the FLACK technique for automatically localizing Alloy buggy expressions causing assertion violations. Given an Alloy model and properties specified by assertions, FLACK first queries the Alloy Analyzer for a counterexample, an instance of the model that does not satisfy the property. Next, FLACK uses a partial max sat (PMAXSAT) solver to find an instance that does satisfy the property and is as close as possible to the counterexample. FLACK then determines the relations and atoms that are different between the counterexample and the satisfying instance. These differences explain how the counterexample violates the assertion. Finally, FLACK analyzes these differences to identify possible buggy expressions. Experimental evaluation of FLACK on 157 Alloy models with a wide variety of bugs shows that FLACK efficiently and consistently ranks buggy expressions in the top 2% of the suspicious list results.

FLACK is different than AlloyFL [8]—the only other Alloy fault localization technique currently available—in that AlloyFL relies on unconventional unit tests while FLACK uses assertions that are natural in the development practices in Alloy. Also, instead of statistically analyzing the effects of tests as in AlloyFL, FLACK relies on counterexample and satisfying instances generated by constraint solving, which are the main underlying technology in Alloy.

In this paper, we focus on the demonstration, implementation, and usage of FLACK. FLACK is highly automatic: the user only needs to provide an assertion to specify the expected behaviors, and FLACK automatically analyzes potential assertion violations and returns a ranked list of expressions based on their suspicious level to the violations. The source code and dataset of FLACK are publicly available at [9]. The full details of FLACK are available in the research paper [7].

II. The Alloy Analyzer

Alloy [1] is a declarative language based on first-order logic, with an analysis engine that relies on a SAT solver. To check that an Alloy model conforms to given assertions, the Alloy Analyzer automatically converts the model and assertions into a boolean formula and uses a SAT solver to search for potential counterexamples violating the assertions.

We now introduce key Alloy terminologies and concepts using the address book model in Figure 1. This model first declares three types (`sig`) `Address`, `Name` and `Book`. The type `Book` has two fields `entry` and `listed`, where `entry` is a set of `Name` and `listed` maps `entry` to a set of `Address` and `Name`.

On lines 8– 10, the model defines the function (`fun`) `lookup`, which finds all `Address` and `Name` associated with a `Name` in a `Book`. Next, on lines 11– 15, the model has a `fact` constraint that specifies that each `Book`'s `entry` maps to at most one `Name` or `Address`. The user makes a mistake on line 14 that uses `lone` (at most one) instead of `some` (at least one), which violates the assertion on lines 29– 32. The

```
 1 abstract sig Listing { }
 2 sig Address extends Listing { }
 3 sig Name extends Listing { }
 4 sig Book {
 5   entry: set Name,
 6   listed: entry ->set Listing
 7 }
 8 fun lookup [b:Book, n:Name] : set Listing {
 9   n.^(b.listed)
10 }
11 fact {
12   all b:Book | all n:b.entry |
13 // Fix: replace "lone" with "some".
14     lone b.listed[n]
15 }
16 fact {
17   all b:Book | all n,l:Name |
18     l in lookup[b,n] implies
19       l in b.entry
20 }
21 fact {
22   all b:Book | all n:b.entry |
23     not n in lookup[b,n]
24 }
25 fact {
26   one Book and one Address and #Name = 2
27 }
28
29 assert assert_1 {
30   all b:Book | all n:b.entry |
31     some (lookup[b,n]&Address)
32 }
33 check assert_1
34
35 pred pred_1 {
36   all b:Book | all n:b.entry |
37     some (lookup[b,n]&Address)
38 }
39 run pred_1
```

Fig. 1: Buggy Address Book Model

fact on lines 16– 20 specifies that all names reachable from any name entry in the book are themselves entries; the fact on lines 21– 24 specifies that name entries are acyclic; and the fact on lines 25– 27 specifies that there should be exactly one `Book`, one `Address` and two `Name`s.

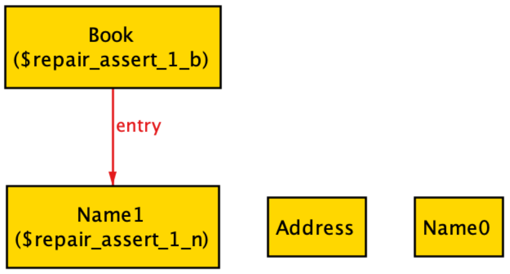

Fig. 2: A counterexample generated by the running `check` command 33.

Alloy uses `assert` to specify assertions and the `check` command to search for counterexamples violating the asserted properties. For example, the assertion on lines 29– 32 specifies that all name entries map to at least one address. This assertion does not always hold, and thus the `check` on line 33 can generate counterexamples showing its violation, e.g., the counterexample in Figure 2 shows an instance of the model in which `Name1` does not map to any `Address`. This violation is caused by the overconstraint on line 14 that uses `lone` (at most one) instead of `some` (at least one). In the next section, we demonstrate how to use FLACK to locate the buggy expression causing this violation.

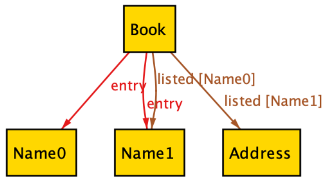

Fig. 3: A satisfying instance.

Alloy uses `pred` to specify predicates and the `run` command to find instances satisfying the predicate properties. For example, the predication on lines 35– 38 specifies the same property that all name entries map to at least one address, and the `run` on line 39 can generate instances that satisfy this property (e.g., the instance in Figure 3 shows an instance in which `Name1` maps to `Address`).

III. FLACK

FLACK is built on top of Alloy 4.2 and is implemented in about 8k LOC in Java. The tool takes as input an Alloy model with assertions and returns a ranked list of suspicious expressions contributing to the violations.

The key insight of FLACK is that the differences between counterexamples and closely related satisfying instances can help identify expressions causing the violation in the input model. To achieve this, FLACK uses a specialized SAT solver to find satisfying instances that are *as close as possible* to the counterexamples (satisfying instances generated by Alloy Analyzer are random and can be vastly different than the counterexample, e.g., the satisfying instance in Figure 3 and the counterexample in Figure 2). Next, FLACK analyzes the differences between the counterexamples and the satisfying instances to find expressions in the model that likely cause the errors. Finally, FLACK computes and returns a ranked list of suspicious expressions.

A. Implementation and Demonstration

Figure 4 gives an overview of the FLACK implementation, which consists of four main phases described below.

Instance generation. This phase is used to obtain pairs of counterexamples and closely similar satisfying instances. FLACK analyzes and compares the differences among these instances to identify likely buggy expressions in the model.

To generate satisfying instances similar to counterexamples, we replace the Alloy's backend SAT solver Kodkod [10]

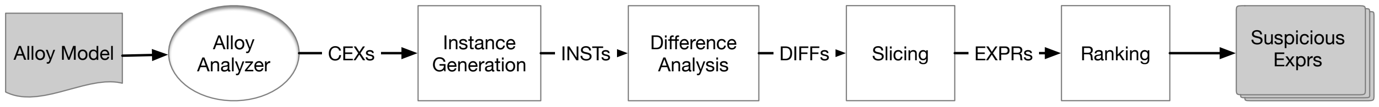

Fig. 4: Workflow of FLACK.

with Pardinus [11], a PMAXSAT (Partial MAXimum SAT-isfiability) solver building on top of Kodkod. We set the Alloy specification with satisfying predicates as the hard constraints and the counterexamples as the soft constraints. The PMAXSAT solver finds a solution that satisfies all hard constraints and maximum number of soft constraints, which results in an instance that satisfies the properties specified by the assertions and most similar to the counterexample.

For the address book model, given the counterexample in Figure 2, FLACK generates the satisfying instance in Figure 5. Notice that this instance is similar to the counterexample in Figure 2, but has an extra edge from `Book` to `Address` labeled with `listed[Name1]`.

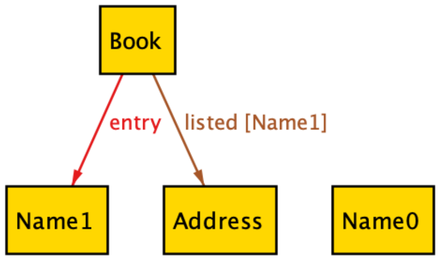

Fig. 5: A satisfying instance that is close to the counterexample in Fig 2.

Difference Analysis. Given pairs of counterexamples and satisfying instances, this phase finds the minimal differences between each pair by comparing the atoms, tuples and relations of each pair. The minimal differences represent the minimal changes needed to convert a counterexample to a satisfying instance, which provide essential information related to the assertion violation.

For the instance pair in Figure 2 and Figure 5, FLACK finds the difference `listed:Book->Name1->Address`, i.e., the counterexample does not contain the tuple `Book->Name1->Address` in relation `listed`. In other words, the assertion violation is most related to the relation `listed` and three atoms: `Book`, `Name1` and `Address`. FLACK uses this information to identify the errors in the following steps.

Slicing. This phase selects expressions most related to the error by collecting expressions that contain the related relations and filtering out other expressions that only contain unrelated relations.

In the address book example, all expressions in `fun` and `facts` on lines 11– 24 are collected, as they all contain the related relation `listed`. The expressions in `fact` on lines 25– 27 are sliced out as it does not contain the relation `listed`, thus are not closely related to the error.

Ranking. The step computes a suspicious score for all collected expressions and their subexpressions and outputs a ranking list, as shown in Figure 6. For each expression, FLACK computes the score by iteratively calculating the scores of all the subexpressions. The score is computed by first instantiating each expression using the values from the differences, then evaluating the instantiated expressions in both counterexamples and satisfying instances, and finally computing the score based on the differences between the evaluated results. Finally, FLACK outputs a ranking list of all expressions and their subexpressions based on the calculated scores.

It is worth noting that our implementation of FLACK does not use the standard Alloy's AST, which is designed for generating instances and is hard to modify. Instead, we extend the original Alloy AST with visitors to traverse and modify AST nodes, so that we can analyze and instrument expressions with concrete values (as used in the Slicing and Ranking phases above).

IV. Usage

FLACK can be used through the command line on operating systems that supports Java. The Alloy Analyzer itself is also a Java application; thus, FLACK and the Alloy Analyzer can be used in the same development environment.

We demonstrate FLACK's usage using the address book example in Figure 1. In this example, the user writes the address book model, with a mistake on line 14 that uses `lone` instead of `some`. The user also inserts an assertion and predicate on lines 29– 39 to specify the expected behaviors.

In the root directory of FLACK, the user asks FLACK to locate errors using the command in Listing 1, where the `-cp ./libs/*:path/to/flack` option specifies the path to all dependent libraries and the jar file of flack, the option `-f /path/to/addr.als` tells FLACK the path to the Alloy model, and the option `-m 5` specifies the maximum number of pairs of counterexamples and satisfying instances to generate.

```
java -Djava.library.path=solvers -cp
    ./libs/*:path/to/flack loc -f
    /path/to/addr.als -m 5
```

Listing 1: Command to run FLACK

Results: For this example, FLACK runs in 0.84 seconds on a 2.2 GHz Intel Core i7 CPU with 16 GB memory, and successfully points to the `lone` error by ranking it first among the suspicious expressions. FLACK outputs a ranking list as shown in Figure 6 [1]. `RANK LIST` lists of suspicious

[1]Due to randomness of Alloy analyzer and the back-end solver, the result may be different.

expressions ranked based on their suspicious score. Each line in the ranking list consists of the ranking position and an expression followed by a suspicious score calculated by FLACK. For example, the first line in the ranking list shows that FLACK correctly ranked `lone ((n.(b.listed)))` [2] first with a suspicious score of 1.31.

Additional Output: FLACK also outputs additional information about the execution for further analysis. The `example generation time` in Figure 6 records the time in seconds used by the Alloy analyzer and the PMAXSAT solver to generate all counterexamples and closely similar instances. The `analyze time(sec)` records the total runtime of FLACK.

The `# rel` and # val are the average numbers of different relations and different values among all counterexamples and satisfying instances, respectively. In this address book model, there are averagely one relation and three atom values that are different between counterexamples and satisfying instances. The `# Slice Out` and `# Total AST` are the number of sliced out and total AST nodes. In this example, FLACK sliced 10 AST nodes out of 74 nodes. The `LOC` is a rough calculation of the lines of code in the model.

The `evals` records the total number of expressions instantiated by the different values. In this example, FLACK evaluated a total number of 368 instantiated expressions.

Note that FLACK is highly automated and uses only one main parameter `-m` to specify the number of pairs of counterexamples and satisfying instances generated. Increasing the number usually results in a more accurate ranking result with a longer execution time. In the usage example, FLACK successfully ranks the buggy expression `lone n.(b.listed)` first using the option `-m 5`. However, if we use `-m 1` for this example, an expression unrelated to the error `l in (b.entry)` is ranked first with a runtime of 0.53 seconds.

V. Experiment

We evaluated FLACK on a benchmark consisting of 157 Alloy models with different kinds of bugs [7]. These include the 152 Alloy models with real faults used in the AlloyFL work [8] and 5 other Alloy models used in complex and real-world applications (e.g., surgical robots [12], Android permissions models [6], and Java program modeling and verification [2]).

The experiment results show that FLACK is able to consistently rank buggy expressions in the top 1.9%(average) of the suspicious list. FLACK successfully rank the buggy expressions for 147 out of 157 models, with 91 (58%) ranked in top 1, 38 (24%) ranked top 2 to 5, 10 (6.5%) ranked top 6 to 10, 8 (5%) ranked above top 10 for 6 and 10 (6.5%) not in the ranking list. For most of the models, FLACK finishes the execution under a second, giving the user instant feedback about the errors. The experiment results show that FLACK is general (can accurately locate a wide variety of bugs) and scalable (can handle complex, real-world Alloy models). The experiment details are given in [7] and publicly available at [9].

[2]This is the syntax desugar of `lone b.listed[n]`

```
/flack/benchmark/addr.als:
example generation time:0.525

RANK LIST:
0: lone ((n . (b . listed))) 1.31
1: n in lookup[b,n] 1.30
2: (n . ^((b . listed))) 1.30
3: l in lookup[b,n] 1.30
4: !(n in lookup[b,n]) 1.30
5: l in (b . entry) 1.17
6: l in lookup[b,n] => l in (b.entry) 1.00

-------------------

analyze time(sec): 0.84
# rel: 1
# val: 3
# Slice Out: 10
# Total AST: 74
LOC: 21
evals: 368
===================
```

Fig. 6: FLACK's results obtained for the model in Figure 1.

VI. Conclusion

We present the implementation details and usage of FLACK, a fault localization tool for Alloy based on assertions. The key idea of FLACK is to compute counterexample and satisfying instances of the model and compare their differences to locate errors. FLACK is fully automated and the user only needs to provide assertions to specify expected behaviors. The source code of FLACK, its benchmark models, and experimental results are publicly available at [9].

As can be seen, currently FLACK is a command line tool that the user manually executes to identify faults. In future work, we plan to integrate FLACK directly to the user interface of the Alloy Analyzer (e.g., top ranked expressions and suspicious components will be highlighted in the Alloy UI using different colors). This would allow the Alloy users to visually analyze faults in Alloy models and potentially help them repair errors more effectively (by making changes to the highlighted expressions).

Acknowledgment

We thank the anonymous reviewers for helpful comments. This work was supported in part by awards W911NF-19-1-0054 from the Army Research Office; CCF-1948536, CCF-1755890, CCF-1618132 from the National Science Foundation; and a Faculty Seed Award from UNL; and PICT 2016-1384, 2017-1979 and 2017-2622 from the Argentine National Agency of Scientific and Technological Promotion (ANPCyT).

REFERENCES

[1] D. Jackson, "Alloy: A lightweight object modelling notation," *ACM Trans. Softw. Eng. Methodol.*, 2002.

[2] J. P. Galeotti, N. Rosner, C. G. López Pombo, and M. F. Frias, "Analysis of invariants for efficient bounded verification," in *International Symposium on Software Testing and Analysis*, 2010.

[3] P. Abad, N. Aguirre, V. S. Bengolea, D. Ciolek, M. F. Frias, J. P. Galeotti, T. Maibaum, M. M. Moscato, N. Rosner, and I. Vissani, "Improving test generation under rich contracts by tight bounds and incremental SAT solving," in *International Conference on Software Testing, Verification and Validation*, 2013.

[4] F. A. Maldonado-Lopez, J. Chavarriaga, and Y. Donoso, "Detecting network policy conflicts using alloy," in *International Conference on Abstract State Machines*, 2014.

[5] M. Alhanahnah, C. Stevens, and H. Bagheri, "Scalable analysis of interaction threats in iot systems," in *International Symposium on Software Testing and Analysis*, 2020.

[6] H. Bagheri, A. Sadeghi, J. Garcia, and S. Malek, "Covert: Compositional analysis of android inter-app permission leakage," *IEEE Transactions on Software Engineering*, 2015.

[7] G. Zheng, T. Nguyen, S. Gutiérrez Brida, G. Regis, M. Frias, N. Aguirre, and H. Bagheri, "Flack: Counterexample-guided fault localization for alloy models," in *International Conference on Software Engineering*, 2021.

[8] K. Wang, A. Sullivan, D. Marinov, and S. Khurshid, "Fault Localization for Declarative Models in Alloy," in *International Symposium on Software Reliability Engineering*, 2020.

[9] *FLACK repository*, 2020. [Online]. Available: https://doi.org/10.6084/m9.figshare.13439894.v4

[10] E. Torlak and D. Jackson, "Kodkod: A relational model finder," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2007.

[11] A. Cunha, N. Macedo, and T. Guimarães, "Target oriented relational model finding," in *Fundamental Approaches to Software Engineering*, 2014.

[12] N. Mansoor, J. A. Saddler, B. Silva, H. Bagheri, M. B. Cohen, and S. Farritor, "Modeling and testing a family of surgical robots: An experience report," in *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018.