FLACK: Counterexample-Guided Fault Localization for Alloy Models

Guolong Zheng*, ThanhVu Nguyen*, Simón Gutiérrez Brida†, Germán Regis†,
Marcelo F. Frias‡, Nazareno Aguirre†, Hamid Bagheri*
*Univeristy of Nebraska-Lincoln
{gzheng, tnguyen}@cse.unl.edu, bagheri@unl.edu
†University of Rio Cuarto and CONICET
{sgutierrez, gregis, naguirre}@dc.exa.unrc.edu.ar
‡Dept. of Software Engineering Instituto Tecnológico de Buenos Aires
mfrias@itba.edu.ar

Abstract—**Fault localization is a practical research topic that helps developers identify code locations that might cause bugs in a program. Most existing fault localization techniques are designed for imperative programs (e.g., C and Java) and rely on analyzing correct and incorrect executions of the program to identify suspicious statements. In this work, we introduce a fault localization approach for models written in a declarative language, where the models are not "executed," but rather converted into a logical formula and solved using backend constraint solvers. We present FLACK, a tool that takes as input an Alloy model consisting of some violated assertion and returns a ranked list of suspicious expressions contributing to the assertion 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. The experimental results show that FLACK is efficient (can handle complex, real-world Alloy models with thousand lines of code within 5 seconds), accurate (can consistently rank buggy expressions in the top 1.9% of the suspicious list), and useful (can often narrow down the error to the exact location within the suspicious expressions).**

I. INTRODUCTION

Declarative specification languages and the corresponding formally precise analysis engines have long been utilized to solve various software engineering problems. The Alloy specification language [1] relies on first-order relational logic, and has been used in a wide range of applications, such as program verification [2], test case generation [3], [4], software design [5], [6], [7], network security [8], [9], [10], security analysis of emerging platforms, such as IoT [11] and Android [12], [13], and design tradeoff analysis [14], [15], to name a few. Cunha and Macedo, among others, use a recent extension of Alloy, called Electrum [16], to validate the European Rail Traffic Management System, a system of standards for management and inter-operation of signaling for the European railways [17]. Kim [18] proposes a Secure Swarm Toolkit (SST), a platform for building an authorization service infrastructure for IoT systems, and uses Alloy to show that SST provides necessary security guarantees.

Similar to developing programs in an imperative language, such as 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, rendering debugging thereof even more arduous. These challenges call for debugging assistant mechanisms, such as fault localization techniques, that support declarative specification languages.

However, there is a dearth of fault localization techniques developed for Alloy. AlloyFL [19] is perhaps the only fault localization tool available for Alloy as of today. The key idea of AlloyFL is to use "unit tests," where a test is a predicate that describes an Alloy instance to encode expected behaviors, to compute suspicious expressions in an Alloy model that fails these tests. To compute the suspicious expressions, AlloyFL uses mutation testing [20], [21] and statistical debugging techniques [22], [23], [24], i.e., it mutates expressions, collects statistics on how each mutation affects the tests, then uses this information to assign suspicion scores to expressions.

While AlloyFL pioneered fault localization in the Alloy context and the obtained results thereof are promising, it relies on the assumption of the availability of AUnit tests [25]—i.e., predicates representing Alloy instances—which are not common in the Alloy setting. Indeed, instead of writing test cases, Alloy users write assertions to describe the desired property and let the Alloy Analyzer search for potential counterexamples (cex's) that violate the property. Moreover, it is unclear how many test cases are needed or how good they must be for AlloyFL to be effective (e.g., in the AlloyFL evaluation [19], the number of tests ranges from 30 to 120).

To address this state of affairs and improve the quality of Alloy development, we present an automated approach and an accompanying tool-suite for **f**ault **l**ocalization in **A**lloy models using **c**ounterexamples, called FLACK. Given an Alloy model and a property that is not satisfied by the model, FLACK first queries the underlying 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 cex and sat instance. Finally, FLACK analyses these differences to compute suspicion scores for expressions in the original model.

Unlike AlloyFL that relies on unconventional unit tests, FLACK uses well-established and widely-used assertions, naturally compatible with the development practices in Alloy. Also, instead of using mutation testing or statically analyzing the effects of tests, FLACK relies on counterexamples and satisfying instances generated by constraint solvers, which are the main underlying technology in Alloy.

We evaluated FLACK on a benchmark consisting of a suite of buggy models from AlloyFL [19]. The experimental results corroborate that FLACK is able to consistently rank buggy expressions in the top 1.9% of the suspicious list. We also evaluated FLACK on three case studies consisting of larger Alloy models used in the real-world settings (e.g., Alloy model for surgical robots, Java programs and Android permissions), and FLACK was able to identify the buggy expressions within the top 1%. The run time of FLACK for most the models is under 5 seconds (under 1 second for the AlloyFL benchmarks). The experimental results corroborate that FLACK has the potential to facilitate a non-trivial task of formal specification development significantly and exposes opportunities for researchers to exploit new debugging techniques for Alloy.

To summarize, this paper makes the following contributions:

- *Fault localization approach for declarative models*: We present a novel fault localization approach for declarative models specified in the Alloy language. The insight underlying our approach is that expressions in an Alloy model that likely cause an assertion violation can be identified by analyzing the counterexamples and closely related satisfying instances.
- *Tool implementation*: We develop a fully automated technology, dubbed FLACK, that effectively realizes our fault localization approach. We make FLACK publicly available to the research and education community [26].
- *Empirical evaluation*: We evaluate FLACK in the context of faulty Alloy specifications found in prior work and specifications derived from real-world systems, corroborating FLACK's ability to consistently rank buggy expressions high on the suspicious list, and analyze complex, real-world Alloy models with thousand lines of code.

The rest of the paper is organized as follows. Section II motivates our research through an illustrative example. Section III describes the details of our fault localization approach for Alloy models. Section IV presents the implementation and evaluation of the research. The paper concludes with an outline of the related research and future work.

II. ILLUSTRATION

To motivate the research and illustrate our approach, we provide an Alloy specification of a finite state machine (FSM), adapted from AlloyFL benchmarks [19]. The specification defines two type signatures, i.e., `State` and `FSM`, along with their fields (lines 1–5). The specification contains three fact paragraphs, expressing the constraints, detailed below: If a start (or a stop) state exists, there is only one of them (fact `OneStartAndStop`). The start state is not a subset of the stop state; no transition terminates at the start state; and no transition leaves a stop state (fact `ValidStartAndStop`). Finally, every state is reachable from the start state, and the stop state is reachable from any state (fact `Reachability`).

```
1  one sig FSM {
2    start: set State,
3    stop: set State
4  }
5  sig State { transition: set State }
6  fact OneStartAndStop {
7    // If a start state exists, there is only one of them
8    all start1, start2 : FSM.start | start1 = start2
9    // If a stop state exists, there is only one of them
10   all stop1, stop2 : FSM.stop | stop1 = stop2
11   some FSM.stop
12 }
13 fact ValidStartAndStop {
14   // start state is not a subset of stop state
15   FSM.start !in FSM.stop
16   // No transition ends at the start state.
17   all s : State | FSM.start !in s.transition
18   // Error: should be "<=>" instead of "=>".
19   all s: State | s.transition = none => s in FSM.stop
20 }
21 fact Reachability {
22   // All states are reachable from the start state.
23   State = FSM.start.*transition
24   // The stop state is reachable from any state.
25   all s: State | FSM.stop in s.*transition
26 }
27 assert NoStopTransition{
28   no FSM.stop.transition
29 }
30 check NoStopTransition for 5
```

Fig. 1: Buggy FSM model, adapted from AlloyFL [19].

Each `assertion` specifies a property that is expected to hold in all instances of the model. For example, we use the assertion `NoStopTransition` to check that a stop state behaves as a sink. The Alloy Analyzer disproves this assertion by producing a counterexample, shown in Figure 2a, in which the stop state labeled `State3` transitions to `State1`.

Thus, there is a "bug" in the model causing the assertion violation. Indeed, careful analysis of the model and the generated cex reveals that the problem is in the expression on line 19: instead of stating that a stop state does not have any transition to any state, the expression states that any state not having a transition to anywhere is a stop state—a subtle logical error that is difficult to realize[1].

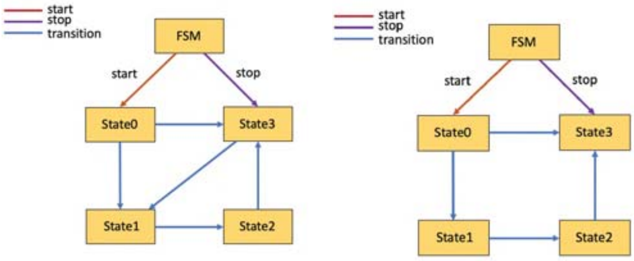
The goal of FLACK is to identify such buggy expressions automatically. For this example, within a second, FLACK identifies four suspicious expressions with the one on line 19 ranked first. Table I shows the results: expressions or nodes with higher scores are ranked higher. Moreover, FLACK suggests that the operator => is likely the issue in the expression.

---

[1] There are two potential fixes for this: (i) reverse the expression to: `s in FSM.stop => s.transition=none` or (ii) replace the implication operator (=>) to logical equivalence (<=>), which technically would strengthen the intended requirement.

TABLE I: FLACK's results obtained for the model in Figure 1.

Suspicious Expression	Score
s.transition = none => s in FSM.stop (=>) 19	1.58
s.transition = none 19	1.25
FSM.stop in s.*transtion 25	0.5
s in FSM.stop 19	0.5



(a) Counterexample (b) (PMAX) Sat instance

Fig. 2: Instance Pair. Note the similarity between the instances.

Such a level of granularity can significantly help the developer understand and fix the problem. The results in Section IV show that FLACK can consistently rank the exact buggy expression within the top 5 suspicious ones and do so under a second.

The key idea underlying our fault localization approach is to analyze the differences between *counterexamples* (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. FLACK first checks the assertion NoStopTransition in the model using the Alloy Analyzer, which returns the cex in Figure 2a. Next, FLACK generates a satisfying (sat) instance that is as minimal and similar to the cex as possible. Their differences promise effective localization of the issue.

a) Generating SAT instances: To obtain an instance similar to the cex, FLACK transforms the input model into a logical formula representing *hard constraints* and the information from cex into a formula representing *soft constraints*. Essentially, FLACK converts the instance finding problem into a *Partial-Max SAT* problem [27] and then uses the Pardinus [28] solver to find a solution that satisfies all the hard constraints and as many soft constraints as possible. Thus, the result is an instance of the model that is similar to the cex but satisfies the assertion. Figure 2b shows an instance produced by Pardinus, considering the cex shown in Figure 2a. Notice that this instance is similar to the given cex, except for the edge from `State3` to `State1`, which represents the main difference between the two instances.

b) Finding Suspicious Expressions: FLACK analyzes the differences between cexs and the sat instances—e.g., here the transition from `State3` to `State1`, which only appears in the cex but not the sat instance—to identify Alloy relations causing the issue. As shown in Table II, that demonstrates the Alloy text representation of the cex in Figure 2a, the `transition` relation involves the tuple `State3 -> State1` and the `stop` relation involves `State3`. Thus, FLACK hypothesizes that two relations of `transition` and `stop` may cause the difference in the two models. Note that while we present one cex and one sat instance in this example for the sake of simplicity, FLACK supports analyzing multiple pairs of cex and sat instances in tandem.

TABLE II: Text Representation of the cex in Figure 2a

Relation	Tuples
FSM	FSM0
start	FSM0->State0
stop	FSM0->State3
State	State0, State1, State2, State3
transition	State0->State1, State0->State3, State1->State2, State2->State3, State3->State1

Next, FLACK *slices* the input model to contain only expressions affecting both relations `transition` and `FSM.stop`. This results in two expressions: `all s: State | FSM.stop in s.*transition` (line 25) and `all s: State | s.transition = none => s in FSM.stop` (line 19).

At this point, FLACK could stop and return these two expressions, one of which is the buggy expression on line 19. Indeed, this level of "statement" granularity is often used in fault localization techniques, like Tarantula [29] or Ochiai [22]. However, FLACK aims to achieve a finer-grained granularity level by also considering the *boolean* and *relational* subexpressions, detailed below.

c) Ranking Boolean Nodes: The expressions on lines 25 and 19 have four boolean nodes: (a) `FSM.stop in s.*transition`, (b) `s.transition = none`, (c) `s in FSM.stop`, and (d) `s.transition = none => s in FSM.stop`. FLACK instantiates each of these with `State1` and `State3`, the values that differentiate the cex and sat instance. For example, (a) becomes `FSM.stop in S1.*transition` and `FSM.stop in S3.*transition`. Next, FLACK evaluates these instantiations using the cex and sat instance and assigns a higher suspicious score to those with inconsistent evaluation results. For example, the instantiations `FSM.stop in S1.*transition` and `FSM.stop in S3.*transition` of node (a) evaluate to `true` in both cex and sat instance, so we give (a) the score 0 (i.e., no changes). We assign score 1 to (b) because `State3.transition = none` evaluates to `false` in the cex but `true` in the sat instance (thus 1 change) and `State1.transition = none` evaluates to false in both (no change).

Overall, FLACK obtains the scores 0, 1, 0, 1 for nodes (a), (b), (c), (d), respectively. Thus, FLACK determines that nodes (b) `s.transition = none` and (d) `s.transition = none => s in FSM.stop` are the two most suspicious boolean subexpressions within the expression on line 19.

d) Ranking Relational Nodes: While subexpression (d) indeed contains the error, it receives the same score as subexpression (a). To achieve more accurate results[2], FLACK further analyzes the involving relations. FLACK instantiates these relations with `State1` and `State3`, assesses them in the context of the cex and sat instances, and assigns scores based on the evaluations. For example, node (d) `s.transition = none => s in FSM.stop` contains 3 relations: (1) `s.transition`, (2) `s`, and (3) `FSM.stop`. Instantiating these relations with `State3` and evaluating them using the cex is as follows: (1) becomes {`State1`}, (2) {`State3`}, and (3) {`State3`}. Thus, for the cex, (d) involves both `State1` and `State3`, and FLACK gives it a score of 1. Next, it evaluates the instantiations using the sat instance as follows: (1) becomes {}, (2) {`State3`}, and (3) {`State3`}. (d) does not involve `State1` and thus has a score 0. FLACK assigns (d) the average score of 0.5 (for the instantiations of `State3`). Performing a similar computation for the instantiation of `State1`, we obtain a score of 2/3 for (d) as the evaluation for the cex and sat instances involves both `State1` and `State3` (differentiated values) and `State2` (regular value). Thus, (d) has a score of 0.58 as an average of 0.5 and 2/3.

Overall, FLACK obtains the scores 0.5, 0.25, 0.5, 0.58 for (a), (b), (c), and (d), respectively. Note that the node (d) is now ranked higher than (a) as desired.

e) Suspicious Scores: FLACK computes the final suspicious score of a node as the sum of the boolean and relational scores of that node, as shown above. For example, the node (d) `s.transition = none => s in FSM.stop` in the expression on line 19 has the highest suspicious score of 1.58. Table I shows suspicious scores of the expressions in the ranked list returned by FLACK.

In addition, FLACK analyzes (non-atomic) nodes containing (boolean) connectors and reports connectors that connect subnodes with different scores. For example, FLACK suggests that the operator => is likely responsible for the error in (d) because the two subexpressions `s.transition = none` and `s in FSM.stop` have different scores as shown in Table I. Indeed, in this example, the assertion violation is entirely due to this operator (a potential fix would be strengthening the model to <=> or switching the two subexpressions as Alloy does not have the operator <=).

III. APPROACH

Figure 3 gives an overview of FLACK, which takes as input an Alloy model with some violated assertion and returns a ranked list of suspicious expressions contributing to the assertion violation. The insight guiding our research is that the differences between counterexamples that do not satisfy the assertion and closely related satisfying instances can drive localization of suspicious expressions in the input model. To achieve this, FLACK uses the *Alloy analyzer* to find counterexamples showing the violation of the assertion. It then uses a PMAX-SAT solver to find satisfying instances that are *as close as possible* to the cex's. Next, FLACK analyzes the differences between the cex's and satisfying instances to find expressions in the model that likely cause the errors. Finally, FLACK computes and returns a ranked list of suspicious expressions.

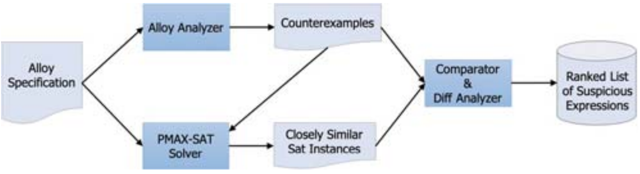

Alloy Specification
Alloy Analyzer
Counterexamples
PMAX-SAT Solver
Closely Similar Sat Instances
Comparator & Diff Analyzer
Ranked List of Suspicious Expressions

Fig. 3: Overview of FLACK.

A. The Alloy Analyzer

An Alloy specification or *model* consists of three components: (i) Type signatures (`sig`) define essential *data types* and their *fields* capture relationships among such data types, (ii) `facts`, predicates (`pred`), and assertions (`assert`) are formulae defining constraints over data types, and (iii) `run` and `check` are commands to invoke the Alloy Analyzer. The `check` command is used to find counterexamples violating some asserted property, and `run` finds satisfying model instances (*sat instances*). For a model M and a property p, a cex is an instance of M that satisfies $M \land \neg p$, and a sat instance is one that satisfies $M \land p$. The specification in Figure 1 defines two signatures (`FSM, State`), three fields (`start, stop, transition`), three facts (`OneStartStop, ValidStart, ValidStop`) and one assertion (`NoStopTransition`).

Analysis of specifications written in Alloy is entirely automated, yet bounded up to user-specified scopes on the size of type signatures. More precisely, to check that p is satisfied by *all* instances of M (i.e., p is valid) up to a certain scope, the Alloy developer encodes p as an *assertion* and uses the `check` command to validate the assertion, i.e., showing that no cex exists within the specified scope (a cex is an instance I such that $I \models M \land \neg p$). To check that p is satisfied by *some* instances of M, the Alloy developer encodes p as a predicate and uses the `run` command to analyze the predicate, i.e., searching for a sat instance I such that $I \models M \land p$. In our running example, the `check` command examines the `NoStopTransition` assertion and returns a cex in Figure 2a.

Internally, Alloy converts these tasks of searching for instances into boolean formulae and uses a SAT solver to check the satisfiability of the formulae. Each value of each relation is translated to a distinct variable in the boolean formula. For example, given a scope of 5 in the FSM model in Figure 1, the relation `State` contains 5 values and is translated to 5 distinct variables in the boolean formula, and the `transition` is translated to 25 values representing 25 values of combinations of $\|State\| \times \|State\|$. An instance is an assignment for all variables that makes the formula True. For example, *cex* in Figure 2a is an assignment where all variables corresponding to values in Table I are assigned True and others are False. Finally, Alloy translates the result from the SAT solver, e.g.,

---

[2] While this example has only two expressions with similar scores, we obtain many expressions with similar scores in more complex and real-world models. Thus, this step is crucial to distinguish the buggy expressions from the rest.

Algorithm 1: FLACK fault localization process

```
input  : Alloy model $M$, property $p$ not satisfied by $M$
output : Ranked list of suspicious expressions in $M$
AlloySolver ← AlloyAnalyzer($M, p$)
pairs ← $\emptyset$
while |pairs| < max_instance_pairs do
    c ← AlloySolver.gencex()
    AlloySolver.blockcex(c)
    s ← PMaxSolver($M$, c)
    if s = nil then
        $U$ ← AlloySolver.get_unsatcore()
        return unsat_analyzer($M, U$, c)
    pairs ← pairs ∪ (c, s)
end
diffs ← comparator(pairs)
return diffs_analyzer(diffs)
```

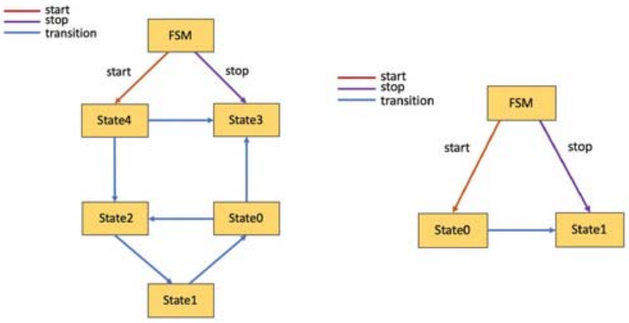


Fig. 4: Model instances generated by the Alloy Analyzer.

an assignment that makes the boolean formula True, back to an instance of M.

B. *The* FLACK *Algorithm*

Algorithm 1 shows the algorithm of FLACK, which takes as input an Alloy model M and a property p that is not satisfied by M (as an assertion violation) and returns a ranked list of expressions that likely contribute to the assertion violation. FLACK first uses the Alloy Analyzer and the Pardinus PMAX-SAT solver to generate pairs of cex and closely similar sat instances. FLACK then analyzes the differences between the cex and sat instances to locate the error. If FLACK cannot generate any sat instance, FLACK inspects the *unsat core* returned by the Alloy Analyzer to locate the error.

1) Generating Instances: To understand why M does not satisfy p, FLACK obtains differences between cexs and relevant sat instances. These differences can lead to the cause of the error. One option is to use the Alloy Analyzer to generate a sat instance directly (e.g., by checking a *predicate* consisting of p). However, such an instance generated by Alloy is often predominantly different from the cex, and thus does not help identify the main difference. For example, the cex, shown in Figure 2a, that violates the assertion `NoStopTransition` is quite different from the two Alloy-generated satisfying instances, shown in Figure 4.

To generate a sat instance closely similar to the cex, we reduce the problem to a PMAX-SAT (partial maximum satisfiability) problem.

Definition III.1 (Finding a Similar Sat Instance from a Cex)**.** Given a set of *hard* clauses, collectively specified by model M and property p, and a set of *soft* clauses, specified by a counterexample cex, find a solution that satisfies *all* hard clauses and satisfies *as many soft clauses as possible.*

More specifically, the hard clauses are generated by constraints in the Alloy model, and the soft clauses are the assignment represented by *cex* where all presenting variables are True and other variables are False. Because the relations and scope of the Alloy model stay the same, the variables in the transformed boolean formula would also remain the same, and just the values assigned to them would differ between various model instances. Thus, this encoding can apply to general instances regardless of their structures.

FLACK then uses an existing PMAX-SAT solver (Pardinus) to find an instance that has the property p and is as similar to the cex as possible[3]. For example, the instance in Figure 2b generated by Pardinus is similar to the cex in Figure 2a, but has an extra edge from `State3` to `State1`. The idea is that such (minimal) differences can help FLACK identify the error.

2) Comparator: FLACK compares the generated cex's and sat instances to obtain their differences, which involve atoms, tuples, and relations. First, it obtains tuples and their atoms that are different between the cex and sat instance, e.g., in Figure 2, the tuple `State3->State1`, which has the atoms `State1` and `State3`, is in the cex but not in the satisfying instance. Next, it obtains relations with different tuples between the cex and sat instance, e.g., the `transition` relation involves the tuple `State3->State1` in the cex but not in the sat instance. Third, it obtains relations that can be inferred from the tuples and atoms derived in the previous steps, e.g., the relation `FSM.stop` involves tuples having the `State3` atom.

In summary, for the pair of cex and sat instance in Figure 2, FLACK obtains the suspicious relations `transitions` and `stop` and the atoms `State1, State3`. FLACK applies these comparison steps for all pairs of instances and cex's and uses the common results.

3) Diff Analyzer: After obtaining the differences consisting of relations and atoms between cex's and sat instances, FLACK analyzes them to obtain a ranked list of expressions based on their suspicious levels. FLACK assigns higher suspicious scores to expressions whose evaluations depend on these differences (and lower scores to those not depending on these differences).

Algorithm 2 shows the Diff Analyzer algorithm, which takes as input a model M, the differences diffs obtained in Sect III-B2, and pairs of cex and sat instances obtained in Sect III-B1, and outputs a ranked list of suspicious expressions in M. It first identifies expressions in M that involve relations in diffs. These expressions are likely related to the difference between cex and sat instance. For example, consider the model in Figure 1. FLACK identifies two expressions:

---

[3] Based on our experiment, the first solution returned by the PMAX-SAT solver is similar enough for FLACK to locate bugs.

Algorithm 2: Diff Analyzer

```
input  : Alloy model $M$, differences diffs, pairs of cex's and sat
         instances pairs
output : Ranked list of suspicious expressions in $M$
exprs ← get_susp_exprs($M$, diffs)
results ← {}
foreach expr ∈ exprs do
    computescore (expr, results)
end
return sort (results)
Function computescore (expr, results) :
    score = 0
    if isleaf (expr) then
        isexpr ← instantiate(expr, diff)
        foreach (c, s) ∈ pairs do
            cvals ← eval(c, isexpr)
            svals ← eval(s, isexpr)
            instscore ← 0
            if diff ⊂ cvals then
                instscore ← instscore + $\frac{|diff|}{|cvals|}$
            if diff ⊂ svals then
                instscore ← instscore + $\frac{|diff|}{|svals|}$
            score ← score + $\frac{instscore}{2}$)
        end
        score ← $\frac{score}{|pairs|}$
    else
        if isbool(expr) then
            isexpr ← instantiate(expr, diff)
            foreach (c, s) ∈ pairs do
                if eval(c, isexpr) ≠ eval(s, isexpr) then
                    score ← score + 1
            end
        foreach child ∈ getchildren(expr) do
            score ← score + computescore(child, results)
        end
    end
    results ← results ∪ (expr, score)
    return score
```

`all s: State | FSM.stop in s.*transition` on line 25 and `all s: State | s.transition = none => s in FSM.stop` on line 19, as they involve the relations `transition` and `stop` in diffs.

FLACK then recursively computes the suspicious score for each collected expression e, represented as an AST tree. If e is a leaf (e.g., a relational expression), FLACK instantiates e with atoms from diffs. FLACK then evaluates the instantiated expression for each pair of cex and sat instance. If the evaluated result for an instantiated expression contains *all* atoms involved in diffs, FLACK computes the score as "*size of* diffs */ size of evaluated results*;" otherwise, the score is 0. For a pair, the score is then the average score of cex and sat instance. At last, the score of e is the average among all pairs. Essentially, a higher suspicious score is assigned to a relational subexpression whose evaluation involves many atoms in diffs.

If e is not a leaf node, e's score is the sum of boolean and relational scores. If e is a boolean expression (i.e., an expression that returns `true` or `false`), we instantiate e with atoms from diffs and evaluate it on each cex and sat instance pair. If it evaluates to different results between the cex and sat instance (e.g., one is `true` and the other is `false`), FLACK increases e's score by 1. Thus, a higher boolean score is assigned to the expressions whose evaluation does not match between pairs of the cex and sat instances. Then e's relational score is calculated as the sum of e's children. The final score assigned to each expression is the sum of the e's boolean scores and the relational scores of e's children. In the end, FLACK returns all expression ranked by their suspicious scores.

To make the idea concrete, consider the expression `s.transition = none` in Figure 1. For the cex and sat instance pair in Figure 2, diffs contain two atoms `State1` and `State3`. FLACK first instantiates the expression under analysis with the atoms mentioned above into two concrete expressions: (1) `State1.transition = none` and (2) `State3.transition = none`. The concrete expression (1) evaluates to `false` in both cex and sat instance, while the concrete expression (2) evaluates to `true` in cex and `false` in the sat instance. Thus, the boolean score for the expression under analysis is 1 as the aggregation of the values obtained for the concrete expressions (1) and (2).

FLACK then computes the relational score for the expression under analysis as the sum of the relational scores for its children: `s.transition` and `none`, both of which are leaves. To compute the score for `s.transtion`, it is instantiated to `State1.transition` and `State3.transtion`. `State1.transition` evaluates to `State2` in both cex and sat instance. Thus, it gets a score of 0. For `State3.transition`, in cex, it evaluates to `State1` and gets a score of 1 as the size of different values {`State3, State1`} divided by the size of the instantiated values {`State3`} and the evaluated values {`State1`}. In sat instance, it evaluates to an empty set and gets a score of 0. Overall, `s.transition` gets a relational score of 0.25 as the average of all its instantiated expressions: `State1.transition` (0) and `State3.transition` (0.5). Finally, the overall score of 1.25 is assigned to the expression `s.transition = none` as the aggregation of its boolean and relational scores.

4) UNSAT Core Analyzer: It is possible that we can only generate cex's, but no sat instances, indicating that some constraints in the model have conflicts with the property we want to check. To identify these constraints, FLACK inspects the *unsat core* returned by the Alloy Analyzer. The unsat core explains why a set of constraints cannot be satisfied by giving a minimal subset of conflicting constraints. Those conflicting constraints can help FLACK identify suspicious expressions.

Algorithm 3 outlines the process underlying our UNSAT core analyzer, which takes as input a model M, an unsat core U, and a cex c showing M does not satisfy a property p, and outputs a list of expressions in M conflicting with p. Recall that these values, M, U, and c, are earlier inferred by FLACK as outlined in Algorithm 1.

FLACK starts by producing a sliced model M', in which all expressions in the unsat core are omitted from the original model M. Removing these conflicting expressions would allow us to obtain sat instances from the new model M' to compare with the cex. FLACK now generates a minimal sat instance from M' and compares it with the input cex to

```
Algorithm 3: UNSAT Analyzer
input  : Alloy model $M$, unsat core $U$, counterexample $c$
output : a set of expressions in $M$
$M' \leftarrow$ slice($M, U$)
s ← PMaxSolver($M'$, c)
diffs ← comparator({(c, s)})
exprs ← collect_exprs($U$)
conflicts ← ∅
foreach expr ∈ exprs do
    foreach diff ∈ diffs do
        if eval(expr, diff, $M'$) = false then
            conflicts ← conflicts ∪ expr
    end
end
return conflicts
```

obtain the differences between the cex and the sat instance as shown in Section III-B2. Then, FLACK attempts to identify which of the removed expressions really conflict with p by evaluating them on obtained differences. The idea is that if an expression evaluates to true, then adding that expression back to the model would still allow the sat instance to be generated, i.e., that expression is not conflicting with p. Thus, expressions that evaluate to `false` are ones conflicting with p and are returned as suspicious expressions. Note that we assign similar scores to these resulting expressions because they all contribute to the unsatisfiability of the original model and the intended property.

For example, if we change line 17 in the model shown in Figure 1 to `all s: State | s.transition !in FSM.start`, Alloy would find counterexamples such as the one in Figure 2a, but fail to generate any sat instances. This is because the modified line forces all states to have some transitions, which conflicts with the constraint requiring no transition for stop states.

From the unsat core, FLACK identifies four expressions in the model: (a) `all start1, start2 : FSM.start | start1 = start2`, (b) `some FSM.stop`, (c) `FSM.start !in FSM.stop` and (d) `all s: State | s.transition !in FSM.start`. After removing these four expressions from the model, FLACK can now generate the same sat instance in Figure 2b. As before, the main difference between the cex and sat instance involves two values: `State1` and `State3`. Then, FLACK evaluates each expression using these values. Expressions (a), (b), and (c) are evaluated to `true` for both values, while expression (d) evaluates to `false` for `State3`. Thus, FLACK correctly identifies (d) as the suspicious expression.

IV. EVALUATION

FLACK is implemented in Java 8 and uses Alloy 4.2. We extend the backend KodKod solver [30] in Alloy to use the Pardinus solver [28] to obtain sat instances similar to counterexamples. We also modify the AST expression representation in Alloy to collect and assign suspicious scores to boolean and relational subexpressions.

Our evaluation addresses the following research questions:

- **RQ1**: Can FLACK effectively find suspicious expressions?
- **RQ2**: How does FLACK scale to large, complex models?
- **RQ3**: How does FLACK compare to AlloyFL?

All experiments described below were performed on a Macbook with 2.2 GHZ i7 CPU and 16 GB of RAM.

A. RQ1: Effectiveness

To investigate the effectiveness of FLACK, we use the benchmark models from AlloyFL [19]. Table III shows 152 buggy models collected from 12 Alloy models in AlloyFL. These are real faults collected from Alloy release 4.1, Amalgam [31], and Alloy homework solutions from graduate students. Briefly, these models are *addr* (address book) and *farmer* (farmer cross-river puzzle) from Alloy; *bempl* (bad employer), *grade* (grade book) and *other* (access-control specifications) are from Amalgam [31]; and *arr* (array), *bst* (balanced search tree), *ctree* (colored tree), *cd* (class diagram), *dll* (doubly linked list), *fsm* (finite state machine), and *ssl* (sorted singly linked list) are homework from AlloyFL.

For models with assertions (e.g., from Amalgam [31]), we use those assertions for the experiments. For models that do not have assertions (e.g., homework assignments), we manually create assertions and expected predicates by examining the correct versions or suggested fixes (provided by [19]). Moreover, from the correct models or suggested fixes, we know which expressions contain errors and therefore use them as *ground truths* to compare against FLACK's results. FLACK deals with models containing multiple violated assertions by analyzing them separately and returning a ranked list for each assertion. For illustration purposes, we simulate this by simply splitting models with separate violations into separate models (e.g., `bst2` contains two assertion violations and thus are split into two models `bst2, bst2_1`). Finally, FLACK is highly automatic and has just one user-configurable option: the number of pairs of cex and satisfying instances (which by default is set to 5 based on our experiences).

a) Results: Table III shows FLACK's results. For each model, we list the name, lines of code, the number of nodes that FLACK determined irrelevant and sliced out, and the number of total AST expression nodes. The last two columns show FLACK's resulting ranking of the correct node and its total run time in second. The 28 italicized *models* contain predicate violations, while the other 124 models contain assertion violations. FLACK automatically determines the violation type and switches to the appropriate technique (e.g., using comparator for assertion errors and the unsat analyzer for predicate violations (Section III). Finally, the models are listed in sorted order based on their ranking results.

In summary, FLACK was able to rank the buggy expressions in the top 1 (e.g., the buggy expression is ranked first) for 91 (60%), top 2 to 5 for 35 (23%), top 6 to 10 for 10 (34%), above top 10 for 6 (4%) out of 152 models. For 10 models, FLACK was not able to identify the cause of the errors (e.g., the buggy expression are not in the ranking list), many of which are beyond the reach of FLACK (e.g., the assertion error is not due to any existing expressions in the model, but rather because the model is "missing" some constraints). Finally, regardless

TABLE III: Results of FLACK on 152 Alloy models. Results are sorted based on ranking accuracy. Times are in seconds.

model	loc	total	sliced	rank	time
top 1 (91)	**41**	**120**	**95**	**1**	**0.2**
addr	21	74	10	1	0.6
arr3	24	48	9	1	0.1
arr4	24	64	61	1	0.2
arr5	24	62	59	1	0.2
arr6	25	56	30	1	0.3
arr7	25	63	50	1	0.1
bst2	56	134	56	1	0.4
bst2_1	56	134	95	1	0.3
bst3_2	55	141	68	1	0.2
cd1	27	44	33	1	0.0
cd1_1	27	44	31	1	0.0
cd2	27	35	25	1	0.0
cd3	26	43	32	1	0.0
cd3_1	26	46	31	1	0.0
dll1	37	77	63	1	0.1
dll2	42	77	63	1	0.1
dll3	37	80	59	1	0.0
dll3_1	37	75	49	1	0.1
dll4	37	81	67	1	0.1
dll5_1	39	102	80	1	0.1
dll6	36	113	94	1	0.1
dll7_1	36	73	59	1	0.1
dll8	36	96	76	1	0.1
dll9	38	100	91	1	0.1
dll11	36	87	68	1	0.1
dll12	36	77	63	1	0.1
dll13	36	60	51	1	0.0
dll14_1	37	85	71	1	0.1
dll15	40	126	107	1	0.1
dll16	36	82	68	1	0.1
dll17_1	36	77	63	1	0.1
dll18	36	102	84	1	0.0
dll18_1	36	101	67	1	0.1
dll20	36	90	64	1	0.0
farmer	99	124	30	1	1.6
fsm1_1	30	90	19	1	0.0
fsm7_1	29	59	14	1	0.0
fsm9_4	30	91	78	1	0.1
grade	33	22	7	1	0.0
ssl1	40	168	126	1	0.1
ssl2	40	150	122	1	0.1
ssl3	45	188	135	1	0.0
ssl3_1	44	184	121	1	0.2
ssl4	40	146	118	1	0.1
ssl5	42	188	160	1	0.6
ssl6	42	157	148	1	0.7
ssl6_1	42	157	148	1	0.5
ssl6_2	41	152	119	1	0.1
ssl6_3	42	152	107	1	0.1
ssl7	41	136	95	1	0.1
ssl7_1	40	135	110	1	0.1
ssl8	43	166	119	1	0.1
ssl10	43	155	110	1	0.1
ssl12	40	157	114	1	0.1
ssl14	44	158	149	1	0.5
ssl14_1	44	158	149	1	0.5
ssl14_2	43	153	120	1	0.0
ssl14_3	44	153	108	1	0.1
ssl17	41	152	119	1	0.0
ssl17_1	42	152	106	1	0.1
ssl18_1	40	160	118	1	0.1
ssl18_2	49	160	85	1	0.3
ssl19	40	169	141	1	0.1
arr1	24	45	31	1	0.1
arr2	25	60	47	1	0.2
arr10	24	59	45	1	0.0
bst1	51	171	155	1	0.2
bst4_1	52	163	147	1	0.1
bst5	52	184	168	1	0.1
bst7	52	159	143	1	0.1
bst8_2	54	156	140	1	0.1
bst9	55	185	169	1	0.1
bst10	47	157	140	1	0.6
bst10_2	52	172	156	1	0.1
bst11_1	60	214	198	1	0.1
bst13	53	200	184	1	0.1
bst14	59	202	186	1	0.1
bst15	53	197	181	1	0.1
bst17_1	52	201	185	1	0.1
bst18_1	56	204	188	1	0.1
bst20	52	169	153	1	0.1
bst21	52	182	166	1	0.2
bst22	50	213	158	1	0.6
dll7	38	90	79	1	0.1
dll10	40	91	85	1	0.2
dll14	39	102	91	1	0.1
dll17	37	89	83	1	0.1
fsm1	31	90	71	1	0.0
fsm3	60	67	56	1	0.7
fsm9	30	91	79	1	0.5
fsm9_3	32	91	78	1	0.1
top 2-5 (35)	**40**	**120**	**66**	**2.9**	**0.2**
arr7_1	24	46	9	2	0.9
arr9	27	83	43	2	0.2
bst2_3	52	133	68	2	0.1
bst12	47	146	94	2	0.2
bst18	51	187	115	2	1.4
bst19	47	158	112	2	0.1
bst19_1	52	175	112	2	0.1
dll2_1	42	82	57	2	0.0
dll17_2	36	82	57	2	0.0
dll18_3	36	103	78	2	0.0
dll19	36	83	63	2	0.1
dll20_1	36	88	68	2	0.1
dll20_2	36	88	47	2	0.0
fsm6	29	98	17	2	0.0
fsm9_2	29	90	18	2	0.1
ssl11	42	177	127	2	0.1
bst8	59	134	57	3	0.3
bst8_1	59	134	57	3	0.2
bst22_1	49	199	124	3	0.1
dll1_1	38	86	57	3	0.1
dll18_2	36	107	71	3	0.6
fsm4	31	141	39	3	0.0
fsm5_2	29	69	17	3	0.0
ssl2_1	44	156	72	3	0.1
ssl13	43	174	123	3	0.1
ssl18	40	162	131	3	0.0
arr8	25	80	15	4	0.5
bst2_2	47	147	92	4	0.1
bst3	57	137	97	4	0.1
fsm2	29	70	14	4	0.0
fsm8	29	71	14	4	0.1
arr11	24	83	41	5	0.1
bst10_3	55	162	75	5	0.3
fsm9_1	29	91	18	5	0.1
ssl15	41	161	106	5	0.1
top 6-10 (10)	**51**	**155**	**73**	**7.6**	**0.2**
bst3_1	57	137	58	6	0.2
bst20_1	55	152	61	6	0.2
fsm7	29	64	14	6	0.1
ssl19_1	50	175	106	7	0.6
bst6	51	140	61	8	0.2
bst12_1	56	164	68	8	0.2
bst19_2	55	154	86	8	0.2
ssl19_2	44	174	81	8	0.3
bst17_2	55	177	79	9	0.2
bst22_2	54	209	118	10	0.1
>10 (6)	**51**	**172**	**80**	**12.7**	**0.3**
bst4_2	51	154	61	11	0.3
bst16	64	181	79	12	0.3
bst17	48	186	113	12	0.2
ssl12_1	44	161	78	12	0.4
ssl9	44	153	72	13	0.1
bst22_3	57	199	74	16	0.5
fail (10)	**42**	**104**	**71**	**-**	**0.1**
bst4	46	145	95	-	0.1
bst11	54	196	135	-	0.2
bempl	51	14	7	-	0.0
ctree	30	49	5	-	0.0
dll3_2	36	75	67	-	0.0
other	34	26	19	-	0.0
ssl16	39	137	119	-	0.0
ssl16_1	39	133	115	-	0.0
ssl16_2	47	136	74	-	0.3
ssl16_3	43	133	71	-	0.2

of whether FLACK succeeds or fails, the tool produces the results almost instantaneously (under a second).

b) Analysis: FLACK was able to locate and rank the buggy expressions in 142/152 models. Many of these bugs are common errors in which the developer did not consider certain corner cases. For example, `stu5` contains the buggy expression `all n : This.header.*link | n.elem <= n.link.elem` that does not allow any node without link (the fix is changing to `all n : This.header.*link | some n.link => n.elem <= n.link.elem`). FLACK successfully recognizes the difference that the last node of the list contains a link to itself in the cex but not in the sat instance, and ranks this expression second; more importantly, it ranks first the subexpression `n.elem <= n.link.elem`, where the fix is actually needed. FLACK also performed especially well on 28 models with violated predicates by analyzing the unsat cores and correctly ranked the buggy expressions first.

For six models `bst4_2`, `bst16`, `bst17`, `bst22_3`, `stu9`, and `stu12_1`, FLACK was not able to place the buggy expression within the top 10 (but still within the top 16). For these models, FLACK obtains differences that are not directly related to the error, but consistently appear in both the cex and

TABLE IV: FLACK's results on large complex models

model	loc	total	sliced	rank	time(s)
surgical robots	200	293	278	2	2.3
android permissions	297	1138	673	2	5.2
sll-contains	5250	5562	5510	3	3.0
count-nodes	3791	5064	2861	18	188.7
remove-nth	5063	6336	3306	12	1265.0

stat instances and therefore confuse FLACK.

FLACK was not able to identify the correct buggy expressions in 10 models, e.g., the resulting ranking list does not contain the buggy expressions. Most of these bugs are beyond the scope of FLACK (and fault localization techniques in general). More specifically, the 9 models `bst4`, `bst11`, `bempl`, `ctree`, `dll3_2`, `ssl16`, `ssl16_1`, `ssl16_2`, and `ssl16_3` have assertion violations due to *missing* constraints in predicates and thus do not contain buggy expressions to be localized. For `other`, FLACK did not find the "ground truth" buggy expression (the buggy expression does not contain the different relation) but ranked first another expression that could also be modified to fix the error.

B. RQ2: Real-world Case Studies

The AlloyFL benchmark contains a wide variety of Alloy models and bugs, but they are relatively small ($\approx$50 LoC). To investigate the scalability of FLACK, we consider additional case studies on larger and more complex Alloy models.

a) Surgical Robots: The study in [32] uses Alloy to model highly-configurable surgical robots to verify a critical *arm movement safety* property: the position of the robot arm is in the same position that the surgeon articulates in the control workspace during the surgery procedure and the surgeon is notified if the arm is pushed outside of its physical range. This property is formulated as an assertion and checked on 15 Alloy models representing 15 types of robot arms using different combinations of hardware and software features. The study found that 5 models violate the property.

Table IV, which has the same format as Table III, lists the results. We use all 5 buggy models (each has about 200 LoC) but list them under one row because they are largely similar and share many common facts and predicates but with different configurable values (one model has a fact that has an AngleLmit set to 3 while another has value 6). The buggy expression is also similar and appears in the same fact. For each model, FLACK ranked the correct buggy expressions in the second place in less than 3 seconds. FLACK returned two suspicious expressions (1) `HapticsDisabled in UsedGeomagicTouch.force` and (2) `some notification : GeomagicTouch.force | notification = HapticsEnabled`. Modifying either expression would fix the issue, e.g., changing `Disable` to `Enabled` in (1) or `Enabled` to `Disabled` in (2).

b) Android Permissions: The COVERT project [33] uses compositional analysis to model the permissions of Android OS and apps to find inter-app communication vulnerabilities. The generated Alloy model used in this work does not contain bugs violating assertions, thus we used the MuAlloy mutation tool [34] to introduce 5 (random) bugs to various predicates in the model: 3 binary operator mutations, 1 unary operator mutation, and 1 variable declaration mutation.

Table IV shows the result. FLACK was able to locate 4 buggy expressions (the unary modification ranked 2nd, the binary operations ranked 3rd, 9th, and 11th), but could not identify the other mutated expression (the variable declaration mutation). However, after manual analysis, we realize that this mutated expression does not contribute to the assertion violation (i.e., FLACK is *correct* in not identifying it as a fault).

c) TACO: The TACO (Translation of Annotated COde) project [2] uses Alloy to verify Java programs with specifications. TACO automatically converts a Java program annotated with invariants to an Alloy model with an assertion. If the Java program contains a bug that violates the annotated invariant, then checking the assertion in the Alloy model would provide a counterexample. We use three different Alloy models with violated assertions representing three real Java programs from TACO [2]: `sll-contains` checks if a particular element exists in a linked list; `count-nodes` counts the number of a list's nodes; and `remove-nth` removes the nth element of a list. These (machine-generated) models are much larger than typical Alloy models ($\approx$5000 to 6000 LoC each).

Table IV shows the results. For `ssl-contains`, FLACK ranked the buggy expression third within 3 seconds. This expression helps us locate an error in the original Java program that skips the list header. The faulty expressions of `remove-nth` and `count-nodes` are ranked 12th and 18th, respectively (which are still quite reasonable given the large, >5000, number of possible locations). Note that these buggy expressions consist of multiple errors (e.g., having 5 buggy nodes), causing FLACK to instantiate and analyze combinations of a large number of subexpressions.

Manual analysis on the identified buggy expressions showed that the original Java programs consist of (single) bugs within loops. TACO performs loop unrolling and thus spreads it into multiple bugs in the corresponding Alloy models.

In summary, we found that FLACK works well on large real-world Alloy models. While coming up with correct fixes for these models remain nontrivial, FLACK can help the developers (or automatic program repair tools) quickly locate buggy expressions, which in turn helps understand (and hopefully repair) the actual errors in original models.

C. RQ3: Comparing to AlloyFL

We compare FLACK with AlloyFL [19], which to the best of our knowledge, is the only Alloy fault localization technique currently available. While both tools compute suspicious statements, they are very different in both assumptions and technical approaches. As discussed in Section V, AlloyFL requires *AUnit tests* [25], provided by the user or generated from the correct model, and adopts existing fault localization techniques in imperative programs, such as mutation testing

TABLE V: Comparision with AlloyFL.

tool	top					avg	
	1	5	10	> 10	failed	rank	time(s)
FLACK	91	126	136	6	10	2.4	0.2
AlloyFL	76	128	137	8	7	3.1	32.4

and spectrum-based fault localization; in contrast, FLACK uses violated assertions and relies on counterexamples.

To apply AlloyFL on the 152 benchmark models, we use the best performance configuration and testsuites described in [19]. Specifically, we use the AlloyFl$_{hybrid}$ algorithm with Ochiai formula and reuse tests in [19] (automatically generated by MuAlloy [34] as described in [19]).

Table V compares the results of FLACK and AlloyFL. The two approaches appear to perform similarly, with FLACK being slightly more accurate. Overall, FLACK outperforms AlloyFL, where on average the buggy expressions ranked 2nd and 3rd by FLACK and AlloyFL, respectively. Also, in top 1 ranking FLACK performs much better compared to AlloyFL (91 over 76 models). Moreover, FLACK is much faster, where the average analysis time is far less than a second for FLACK, it takes over 30 seconds for AlloyFL to analyze the same specifications.

We were not able to apply AlloyFL to the models in Section IV-B because MuAlloy [34], which is used to generate AUnit tests for AlloyFL, does not work with these models (e.g., mostly caused by unhandled Alloy operators). This is not a weakness of AlloyFL, but it suggests that it is not trivial to generate tests from existing Alloy models automatically.

D. Threats to Validity

We assume no fault in data type (sig's) and field declarations, which may limit the usage of FLACK. However, none of the benchmark models we used has bugs at these locations. Moreover, we could always translate constraints for sig and field to facts. For example, `one sig A` could be translated to `sig A; fact{one A}`.

The models in the AlloyFL benchmark are collected from graduate student's homework and relatively small. Thus, they may not represent faulty Alloy models in the real world. We also evaluate FLACK with large Alloy models, written by experienced Alloy developer (the surgical robot models and Android permissions model) or generated by an automatic tool (TACO) and show that FLACK performs well on these models (Section IV-B).

We manually create assertions for models that do not have assertions. Thus, our assertions might be inaccurate and not as intended. However, for other models with assertions (e.g., those in the AlloyFL benchmark and all the case studies), we use those assertions directly and FLACK output similar results.

V. RELATED WORK

FLACK is related to AlloyFL [19], which adopts spectrum-based fault localization [22], [23], [24] and mutation-based techniques [20], [21] from imperative languages. Given AUnit tests [25] labeled with should-pass or should-fail, AlloyFL computes a suspicious score for expression by mutating and giving it a higher score if the mutation increases the number of should-pass tests pass and the number of should-fail tests fail. AlloyFL uses MuAlloy [34] to automatically generate tests. However, MuAlloy requires the correct Alloy model to generate these tests. FLACK does not require tests and instead uses assertions, which are commonly used in Alloy.

The generation of similar instances can be viewed as a model exploration problem [35]. Bordeaux [36] uses Alloy to find pairs of SAT/UNSAT instances with minimum relative distances. In contrast, FLACK reduces the generation of an instance as close as possible to the identified counterexamples into a partial max sat problem and solves it using a PMAX-SAT solver.

Amalgam [31] explains why some tuples of a relation do or do not appear in certain instances. A user would manually select a tuple to add or delete, and Amalgam tries to explain why they can or cannot do so (typically the reason for counterexamples is due to the assertions). FLACK instead automatically identifies why a counterexample fails and finds locations that relate to this violation.

Many fault localization techniques have been developed for imperative languages. Spectrum-based techniques [22], [23], [24], [37], [38], [29], [39], [40] identify faulty statements by comparing passing and failing test executions. SAT-based techniques [41], [42] convert the fault location problem into an SAT problem. Statistic-based methods [43], [44] collect statistical information from test executions to locate errors. Feedback-based techniques [45], [46] interactively locates error by getting feedback from the user. Delta debugging [47], [48] identifies code changes responsible for test failure. There are also works on minimizing differences in inputs based on the assumption that similar inputs would lead to similar runs [49], [50], [51]. Program slicing [52], [53], [54] has also been used to aid debugging.

Model-based diagnosis (MBD) approaches identify faulty components of a system based on abnormal behaviors. Griesmayer [55] applied MBD to localizing fault in imperative programs using model checker. Marques-Silva [56] converted the MBD problem into a MAXSAT problem to find the minimal diagnosis, where the system description is encoded as the hard clauses and the not abnormal predicates as the soft clauses. There has also been another similar line work in pinpointing axioms in description logic [57], [58].

VI. CONCLUSION AND FUTURE WORK

We introduce a new fault localization approach for declarative models written Alloy. Our insight is that Alloy expressions that likely cause an assertion violation can be obtained by analyzing the counterexamples, unsat cores, and satisfying instances from the Alloy Analyzer. We present FLACK, a tool that implements these ideas to compute and rank suspicious expressions causing an assertion violation in an Alloy model. FLACK uses a PMAX-SAT solver to find satisfying instances similar to counterexamples generated by the Alloy Analyzer, analyzes satisfying instances and counterexamples to locate suspicious expressions, analyzes subexpressions to achieve a finer-grain level of localization granularity, and uses unsat cores to help identify conflicting expressions. Preliminary results on existing Alloy benchmarks and large, real-world benchmarks show that FLACK is effective in finding accurate expressions causing errors. We believe that FLACK takes an important step in finding bugs in Alloy and exposes opportunities for researchers to exploit new debugging techniques for Alloy.

Currently, we are improving the accuracy and efficiency of FLACK. Specifically, instead of using a default number of instance pairs, we can search for instances incrementally until the algorithm converges. We are also exploring new approaches to effectively integrate FLACK with automatic Alloy repair techniques. Preliminary results from the recent BeAFix work [59] shows that FLACK accurately identifies faults in Alloy specifications, which in turn helps BeAFix automatically analyze and repair those specifications.

VII. DATA AVAILABILITY

We make FLACK and all research artifacts, models, and experimental data reported in the paper available to the research and education community [26].

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, and CCF-1618132 from the National Science Foundation; 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.*, vol. 11, no. 2, p. 256–290, Apr. 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*, ser. ISSTA '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 25–36.

[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*. IEEE Computer Society, 2013, pp. 21–30.

[4] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek, "Reducing combinatorics in gui testing of android applications," in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 559–570.

[5] H. Bagheri and K. J. Sullivan, "Bottom-up model-driven development," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, D. Notkin, B. H. C. Cheng, and K. Pohl, Eds. IEEE Computer Society, 2013, pp. 1221–1224. [Online]. Available: https://doi.org/10.1109/ICSE.2013.6606683

[6] ——, "Pol: specification-driven synthesis of architectural code frameworks for platform-based applications," in *Generative Programming and Component Engineering, GPCE'12, Dresden, Germany, September 26-28, 2012*, K. Ostermann and W. Binder, Eds. ACM, 2012, pp. 93–102. [Online]. Available: https://doi.org/10.1145/2371401.2371416

[7] H. Bagheri, Y. Song, and K. J. Sullivan, "Architectural style as an independent variable," in *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, C. Pecheur, J. Andrews, and E. D. Nitto, Eds. ACM, 2010, pp. 159–162. [Online]. Available: https://doi.org/10.1145/1858996.1859026

[8] F. A. Maldonado-Lopez, J. Chavarriaga, and Y. Donoso, "Detecting network policy conflicts using alloy," in *Proceedings of the 4th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z - Volume 8477*, ser. ABZ 2014. Berlin, Heidelberg: Springer-Verlag, 2014, p. 314–317.

[9] T. Nelson, C. Barratt, D. J. Dougherty, K. Fisler, and S. Krishnamurthi, "The margrave tool for firewall analysis," in *Proceedings of the 24th International Conference on Large Installation System Administration*, ser. LISA'10. USA: USENIX Association, 2010, p. 1–8.

[10] N. Ruchansky and D. Proserpio, "A (not) nice way to verify the openflow switch specification: Formal modelling of the openflow switch using alloy," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, p. 527–528, Aug. 2013.

[11] M. Alhanahnah, C. Stevens, and H. Bagheri, "Scalable analysis of interaction threats in iot systems," in *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, S. Khurshid and C. S. Pasareanu, Eds. ACM, 2020, pp. 272–285. [Online]. Available: https://doi.org/10.1145/3395363.3397347

[12] H. Bagheri, J. Wang, J. Aerts, and S. Malek, "Efficient, evolutionary security analysis of interacting android apps," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 357–368.

[13] H. Bagheri, E. Kang, S. Malek, and D. Jackson, "A formal approach for detection of security flaws in the Android permission system," *Formal Aspects of Computing*, vol. 30, no. 5, pp. 525–544, 2018. [Online]. Available: https://doi.org/10.1007/s00165-017-0445-z

[14] H. Bagheri, C. Tang, and K. Sullivan, "Trademaker: Automated dynamic analysis of synthesized tradespaces," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 106–116. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568291

[15] H. Bagheri, C. Tang, and K. J. Sullivan, "Automated synthesis and dynamic analysis of tradeoff spaces for object-relational mapping," *IEEE Trans. Software Eng.*, vol. 43, no. 2, pp. 145–163, 2017. [Online]. Available: https://doi.org/10.1109/TSE.2016.2587646

[16] J. Brunel, D. Chemouil, A. Cunha, and N. Macedo, "The electrum analyzer: model checking relational first-order temporal specifications," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, M. Huchard, C. Kästner, and G. Fraser, Eds. ACM, 2018, pp. 884–887. [Online]. Available: https://doi.org/10.1145/3238147.3240475

[17] A. Cunha and N. Macedo, "Validating the hybrid ertms/etcs level 3 concept with electrum," in *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, M. Butler, A. Raschke, T. S. Hoang, and K. Reichl, Eds. Cham: Springer International Publishing, 2018, pp. 307–321.

[18] H. Kim, E. Kang, E. A. Lee, and D. Broman, "A toolkit for construction of authorization service infrastructure for the internet of things," in *2017 IEEE/ACM Second International Conference on Internet-of-Things Design and Implementation (IoTDI)*, 2017, pp. 147–158.

[19] K. Wang, A. Sullivan, D. Marinov, and S. Khurshid, "Fault Localization for Declarative Models in Alloy," in *International Symposium on Software Reliability Engineering (ISSRE)*, 2020, pp. 391–402.

[20] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," 03 2014, pp. 153–162.

[21] M. Papadakis and Y. Le Traon, "Metallaxis-fl: Mutation-based fault localization," *Softw. Test. Verif. Reliab.*, vol. 25, no. 5-7, pp. 605–628, Aug. 2015. [Online]. Available: https://doi.org/10.1002/stvr.1509

[22] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the accuracy of spectrum-based fault localization," in *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, ser. TAICPART-MUTATION '07. USA: IEEE Computer Society, 2007, p. 89–98.

[23] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE '02. New York, NY, USA: Association for Computing Machinery, 2002, pp. 467–477.

[24] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 3, Aug. 2011.

[25] A. Sullivan, K. Wang, and S. Khurshid, "Aunit: A test automation tool for alloy," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, 2018, pp. 398–403.

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

[27] Z. Fu and S. Malik, "On solving the partial max-sat problem," in *Theory and Applications of Satisfiability Testing - SAT 2006*, A. Biere and C. P. Gomes, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 252–265.

[28] A. Cunha, N. Macedo, and T. Guimarães, "Target oriented relational model finding," in *Fundamental Approaches to Software Engineering*, S. Gnesi and A. Rensink, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 17–31.

[29] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 273–282.

[30] E. Torlak and D. Jackson, "Kodkod: A relational model finder," in *Tools and Algorithms for the Construction and Analysis of Systems*, O. Grumberg and M. Huth, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 632–647.

[31] T. Nelson, N. Danas, D. J. Dougherty, and S. Krishnamurthi, "The power of "why" and "why not": enriching scenario exploration with provenance," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, 2017, pp. 106–116.

[32] 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 *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 785–790. [Online]. Available: https://doi.org/10.1145/3236024.3275534

[33] H. Bagheri, A. Sadeghi, J. Garcia, and S. Malek, "Covert: Compositional analysis of android inter-app permission leakage," *IEEE Transactions on Software Engineering*, vol. 41, no. 9, pp. 866–886, 2015.

[34] K. Wang, A. Sullivan, and S. Khurshid, "Mualloy: A mutation testing framework for alloy," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 29–32. [Online]. Available: https://doi.org/10.1145/3183440.3183488

[35] N. Macedo, A. Cunha, and T. Guimarães, "Exploring scenario exploration," in *Fundamental Approaches to Software Engineering*, A. Egyed and I. Schaefer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 301–315.

[36] V. Montaghami and D. Rayside, "Bordeaux: A tool for thinking outside the box," in *Proceedings of the 20th International Conference on Fundamental Approaches to Software Engineering - Volume 10202*. Berlin, Heidelberg: Springer-Verlag, 2017, p. 22–39. [Online]. Available: https://doi.org/10.1007/978-3-662-54494-5_2

[37] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund, "A practical evaluation of spectrum-based fault localization," *J. Syst. Softw.*, vol. 82, no. 11, p. 1780–1792, Nov. 2009.

[38] V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight defect localization for java," in *Proceedings of the 19th European Conference on Object-Oriented Programming*, ser. ECOOP'05. Berlin, Heidelberg: Springer-Verlag, 2005, p. 528–550.

[39] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 15–26. [Online]. Available: https://doi.org/10.1145/1065010.1065014

[40] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. IEEE Press, 2017, p. 609–620. [Online]. Available: https://doi.org/10.1109/ICSE.2017.62

[41] M. Jose and R. Majumdar, "Cause clue clauses: Error localization using maximum satisfiability," *SIGPLAN Not.*, vol. 46, no. 6, pp. 437–446, Jun. 2011. [Online]. Available: https://doi.org/10.1145/1993316.1993550

[42] D. Gopinath, R. N. Zaeem, and S. Khurshid, "Improving the effectiveness of spectra-based fault localization using specifications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2012. New York, NY, USA: Association for Computing Machinery, 2012, p. 40–49. [Online]. Available: https://doi.org/10.1145/2351676.2351683

[43] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "Sober: Statistical model-based bug localization," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 286–295, Sep. 2005. [Online]. Available: https://doi.org/10.1145/1095430.1081753

[44] W. E. Wong, V. Debroy, and D. Xu, "Towards better fault localization: A crosstab-based statistical approach," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 42, no. 3, pp. 378–396, 2012.

[45] Y. Lin, J. Sun, Y. Xue, Y. Liu, and J. Dong, "Feedback-based debugging," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE'17. IEEE Press, 2017, pp. 393–403. [Online]. Available: https://doi.org/10.1109/ICSE.2017.43

[46] X. Li, S. Zhu, M. d'Amorim, and A. Orso, "Enlightened debugging," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 82–92. [Online]. Available: https://doi.org/10.1145/3180155.3180242

[47] B. Ness and V. Ngo, "Regression containment through source change isolation," 09 1997, pp. 616 – 621.

[48] A. Zeller, "Yesterday, my program worked. today, it does not. why?" in *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-7. Berlin, Heidelberg: Springer-Verlag, 1999, pp. 253–267.

[49] T. Reps, T. Ball, M. Das, and J. Larus, "The use of program profiling for software maintenance with applications to the year 2000 problem," in *Software Engineering — ESEC/FSE'97*, M. Jazayeri and H. Schauer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 432–449.

[50] D. B. Whalley, "Automatic isolation of compiler errors," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 5, pp. 1648–1659, Sep. 1994.

[51] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, pp. 183–200, Feb. 2002.

[52] J. R. Lyle and M. Weiser, "Automatic program bug location by program slicing," 1987.

[53] H. Agrawal, J. Horgan, S. London, and W. Wong, "Fault localization using execution slices and dataflow tests," *Proceedings of IEEE Software Reliability Engineering*, 06 1999.

[54] H. Agrawal and J. R. Horgan, "Dynamic program slicing," *SIGPLAN Not.*, vol. 25, no. 6, pp. 246–256, Jun. 1990. [Online]. Available: https://doi.org/10.1145/93548.93576

[55] A. Griesmayer, S. Staber, and R. Bloem, "Fault localization using a model checker," *Softw. Test. Verif. Reliab.*, vol. 20, no. 2, p. 149–173, Jun. 2010.

[56] J. Marques-Silva, M. Janota, A. Ignatiev, and A. Morgado, "Efficient model based diagnosis with maximum satisfiability," in *Proceedings of the 24th International Conference on Artificial Intelligence*, ser. IJCAI'15. AAAI Press, 2015, p. 1966–1972.

[57] F. Baader and R. Peñaloza, "Axiom pinpointing in general tableaux," *J. Log. and Comput.*, vol. 20, no. 1, p. 5–34, Feb. 2010. [Online]. Available: https://doi.org/10.1093/logcom/exn058

[58] F. Baader, R. Peñaloza, and B. Suntisrivaraporn, "Pinpointing in the description logic $\mathcal{EL}^+$," in *KI 2007: Advances in Artificial Intelligence*, J. Hertzberg, M. Beetz, and R. Englert, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 52–67.

[59] S. G. Brida, G. Regis, G. Zheng, H. Bagheri, T. Nguyen, N. Aguirre, and M. F. Frias, "Bounded exhaustive search of alloy specification repairs," in *International Conference on Software Engineering (ICSE)*. IEEE, 2021, p. to appear.