FLACK: Counterexample-Guided Fault Localization for Alloy Models

Guolong Zheng*, ThanhVu Nguyen*, Simón Gutiérrez Brida†, Germán Regis†, Marcelo F. Frias‡, Nazareno Aguirre†, Hamid Bagheri*

*Unversity 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 fault localization in Alloy models using counterexamples, 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 illustrates 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).

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.

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 \(\Rightarrow\) is likely the issue in the expression.

```plaintext
// The stop state is reachable from any state.
all s : State | FSM.stop in s.*transition

// The stop state is reachable from any state.
all s: State | s.transition = none \(\Rightarrow\) s in FSM.stop
```

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

1 There are two potential fixes for this: (i) reverse the expression to: \(s\ in\ FSM\ .\ stop \Rightarrow s\ .\ transition=none\) or (ii) replace the implication operator \(\Rightarrow\) to logical equivalence \(\leftrightarrow\), which technically would strengthen the intended requirement.
TABLE I: FLACK’s results obtained for the model in Figure 1

<table>
<thead>
<tr>
<th>Suspicious Expression</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.transition = none =&gt; s in FSM.stop ((\Rightarrow))</td>
<td>1.58</td>
</tr>
<tr>
<td>s.transition = none</td>
<td>1.25</td>
</tr>
<tr>
<td>FSM.stop in s.*transition</td>
<td>0.5</td>
</tr>
<tr>
<td>s in FSM.stop</td>
<td>0.5</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Relation</th>
<th>Tuples</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSM</td>
<td>FSM0</td>
</tr>
<tr>
<td>start</td>
<td>FSM0-&gt;State0</td>
</tr>
<tr>
<td>stop</td>
<td>FSM0-&gt;State3</td>
</tr>
<tr>
<td>State</td>
<td>State0, State1, State2, State3</td>
</tr>
<tr>
<td>transition</td>
<td>State0-&gt;State1, State0-&gt;State3, State1-&gt;State2, State2-&gt;State3, State3-&gt;State1</td>
</tr>
</tbody>
</table>

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 Statel 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 Statel.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 sub-

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 Pardusin [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 2a shows an instance produced by Pardusin, considering the cex shown in Figure 2a. Notice that this instance is similar to the given cex, except for the edge from State3 to Statel, 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 Statel, 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 -> Statel 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.
pression (a). To achieve more accurate results, 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 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 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.

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 ∧ ¬p, and a sat instance is one that satisfies M ∧ 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 = M ∧ ¬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 = M ∧ 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|| × ||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.,

\[ \text{A. The Alloy Analyzer} \]

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 ∧ ¬p, and a sat instance is one that satisfies M ∧ 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 = M ∧ ¬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 = M ∧ 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|| × ||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.,
Algorithm 1: FLACK fault localization process

\begin{verbatim}
input : Alloy model M, property p not satisfied by M
output : Ranked list of suspicious expressions in M
AlloySolver ← AlloyAnalyzer(M, p)
pairs ← ∅
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)
\end{verbatim}

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 in Sect III-B1, and outputs a ranked list of suspicious expressions.

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 cexs and sat instances.

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:

Based on our experiment, the first solution returned by the PMAX-SAT solver is similar enough for FLACK to locate bugs.
all \( s \): State \( | \) FSM.stop in \( s \).transition on line 25 and all \( s \): State \( | \) s.transition = none \( \Rightarrow \) \( s \) in FSM.stop on line 19, as they involve the relations transition and stop in \( \text{diffs} \).

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 \( \text{diffs} \). 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 \( \text{diffs} \), FLACK computes the score as “size of \( \text{diffs} \)/ 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 \( \text{diffs} \).

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 \( \text{diffs} \) 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 \( s\text{.transition} = \text{none} \) in Figure 1. For the cex and sat instance pair in Figure 2, \( \text{diffs} \) 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 size of values obtained for the concrete expressions (1) and (2).

FLACK computes the relational score for the expression under analysis as the sum of the relational scores for its children: \( s\text{.transition} \) and none, both of which are leaves. To compute the score for \( s\text{.transition} \), it is instantiated to State1.transition and State3.transition. 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, \( s\text{.transition} \) 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 \( s\text{.transition} \) = 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
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 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 is 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.

<table>
<thead>
<tr>
<th>model</th>
<th>loc</th>
<th>total</th>
<th>sliced</th>
<th>rank</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr</td>
<td>41</td>
<td>120</td>
<td>95</td>
<td>1</td>
<td>0.2</td>
</tr>
<tr>
<td>arr3</td>
<td>24</td>
<td>48</td>
<td>9</td>
<td>0.6</td>
<td></td>
</tr>
<tr>
<td>arr4</td>
<td>24</td>
<td>64</td>
<td>61</td>
<td>0.2</td>
<td></td>
</tr>
<tr>
<td>arr5</td>
<td>24</td>
<td>62</td>
<td>59</td>
<td>0.2</td>
<td></td>
</tr>
<tr>
<td>arr7</td>
<td>25</td>
<td>63</td>
<td>50</td>
<td>0.3</td>
<td></td>
</tr>
<tr>
<td>bst2</td>
<td>56</td>
<td>134</td>
<td>56</td>
<td>0.1</td>
<td></td>
</tr>
<tr>
<td>bst2_1</td>
<td>56</td>
<td>134</td>
<td>95</td>
<td>0.3</td>
<td></td>
</tr>
<tr>
<td>bst3_2</td>
<td>55</td>
<td>141</td>
<td>68</td>
<td>0.2</td>
<td></td>
</tr>
<tr>
<td>cd1</td>
<td>27</td>
<td>44</td>
<td>33</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>cd2</td>
<td>27</td>
<td>35</td>
<td>36</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>cd3</td>
<td>26</td>
<td>43</td>
<td>12</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>cd3_1</td>
<td>26</td>
<td>46</td>
<td>31</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>dll1</td>
<td>37</td>
<td>77</td>
<td>63</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td>dll2</td>
<td>42</td>
<td>77</td>
<td>63</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td>dll3</td>
<td>37</td>
<td>80</td>
<td>59</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td>dll1_3</td>
<td>37</td>
<td>75</td>
<td>49</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td>dll1_4</td>
<td>37</td>
<td>81</td>
<td>67</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td>dll5</td>
<td>39</td>
<td>102</td>
<td>60</td>
<td>0.1</td>
<td></td>
</tr>
<tr>
<td>dll6</td>
<td>36</td>
<td>113</td>
<td>94</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td>dll7_1</td>
<td>36</td>
<td>73</td>
<td>59</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td>dll10</td>
<td>36</td>
<td>79</td>
<td>67</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td>dll11</td>
<td>36</td>
<td>87</td>
<td>68</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td>dll12</td>
<td>36</td>
<td>77</td>
<td>63</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td>dll13</td>
<td>36</td>
<td>60</td>
<td>51</td>
<td>0.1</td>
<td></td>
</tr>
<tr>
<td>dll14</td>
<td>37</td>
<td>85</td>
<td>71</td>
<td>0.1</td>
<td></td>
</tr>
<tr>
<td>dll15</td>
<td>40</td>
<td>126</td>
<td>107</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td>dll16</td>
<td>36</td>
<td>82</td>
<td>68</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td>dll17</td>
<td>36</td>
<td>77</td>
<td>63</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td>dll18</td>
<td>36</td>
<td>102</td>
<td>84</td>
<td>0.1</td>
<td></td>
</tr>
<tr>
<td>dll18_1</td>
<td>36</td>
<td>101</td>
<td>67</td>
<td>0.1</td>
<td></td>
</tr>
<tr>
<td>dll20</td>
<td>36</td>
<td>90</td>
<td>64</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td>farmer</td>
<td>99</td>
<td>124</td>
<td>30</td>
<td>0.1</td>
<td>0.0</td>
</tr>
<tr>
<td>fsm1_1</td>
<td>30</td>
<td>90</td>
<td>19</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>fsm7_1</td>
<td>29</td>
<td>59</td>
<td>14</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>fsm9_4</td>
<td>30</td>
<td>91</td>
<td>78</td>
<td>0.1</td>
<td></td>
</tr>
<tr>
<td>grade</td>
<td>33</td>
<td>22</td>
<td>7</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>ssl1</td>
<td>40</td>
<td>168</td>
<td>126</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td>ssl2</td>
<td>40</td>
<td>150</td>
<td>122</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td>ssl3</td>
<td>45</td>
<td>188</td>
<td>135</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>ssl3_1</td>
<td>44</td>
<td>184</td>
<td>121</td>
<td>0.2</td>
<td></td>
</tr>
<tr>
<td>ssl4</td>
<td>40</td>
<td>146</td>
<td>118</td>
<td>0.1</td>
<td></td>
</tr>
<tr>
<td>ssl5</td>
<td>42</td>
<td>188</td>
<td>160</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>ssl6</td>
<td>42</td>
<td>157</td>
<td>148</td>
<td>0.7</td>
<td>0.0</td>
</tr>
<tr>
<td>ssl6_1</td>
<td>42</td>
<td>157</td>
<td>148</td>
<td>0.5</td>
<td>0.0</td>
</tr>
<tr>
<td>ssl7</td>
<td>41</td>
<td>157</td>
<td>119</td>
<td>0.1</td>
<td></td>
</tr>
<tr>
<td>ssl7_1</td>
<td>40</td>
<td>135</td>
<td>110</td>
<td>0.1</td>
<td></td>
</tr>
<tr>
<td>ssl8</td>
<td>43</td>
<td>166</td>
<td>119</td>
<td>0.1</td>
<td></td>
</tr>
</tbody>
</table>

The results almost instantaneously (under a second).

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

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

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, bemp1, ctrees, d113_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 (≈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 [II] 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 AngleLimit set to 3 while another has value 6). The buggy expression is also similar and appears in the same fact. For an AngleLimit set to 3 while another has value 6). The buggy expression is similar and share many common facts and predicates but 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]: ssl-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 (≈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 AuUnit 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.

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]: ssl-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 (≈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.
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 test suites 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 [66] 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,

<table>
<thead>
<tr>
<th>tool</th>
<th>1</th>
<th>5</th>
<th>10</th>
<th>&gt; 10</th>
<th>failed</th>
<th>rank</th>
<th>avg time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLACK</td>
<td>91</td>
<td>126</td>
<td>136</td>
<td>6</td>
<td>10</td>
<td>2.4</td>
<td>0.2</td>
</tr>
<tr>
<td>AlloyFL</td>
<td>76</td>
<td>128</td>
<td>137</td>
<td>8</td>
<td>7</td>
<td>3.1</td>
<td>32.4</td>
</tr>
</tbody>
</table>
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 models 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
