Fault Localization for Declarative Models in Alloy

Kaiyuan Wang
Google Inc
Sunnyvale, CA USA
kaiyuanw@google.com

Allison Sullivan
University of Texas
Arlington, TX USA
allison.sullivan@uta.edu

Darko Marinov
University of Illinois
Urbana, IL USA
marinov@illinois.edu

Sarfraz Khurshid
University of Texas
Austin, TX USA
khurshid@utexas.edu

Abstract—Fault localization is a popular research topic and many techniques have been proposed to locate faults in imperative code, e.g., C and Java. In this paper, we focus on the problem of fault localization for declarative models in Alloy—a first-order relational logic with transitive closure. We introduce AlloyFL_{hy}, the first fault localization technique for faulty Alloy models which leverages multiple test formulas. AlloyFL_{hy} brings the traditional spectrum-based and mutation-based fault localization techniques to Alloy and combines both techniques to locate faults. To measure the effectiveness of AlloyFL_{hy}, we define three distance metrics and use both distance-based and top-k metrics to measure the effectiveness of AlloyFL_{hy} on 90 faulty real models. The results show that AlloyFL_{hy} is substantially more effective than Alloy’s built-in unsat core.

Keywords—Alloy, Fault localization, AlloyFL_{hy}

I. INTRODUCTION

Writing declarative models and specifications has numerous benefits, ranging from automated reasoning and correction of design-level properties before systems are built [18, 28], to automated testing and debugging of their implementations after they are built [34]. However, writing correct declarative models that represent non-trivial properties is not easy, especially for practitioners who are not well-versed with the intricate syntax and semantics of declarative languages. Daniel Jackson, the inventor of Alloy, has pointed out in his ICSE keynote [19] that declarative specifications can be "maddening harder to learn and even harder to debug" and the "unsat core is not enough". This motivates us to develop fault localization techniques for declarative models written in Alloy [18]—a first-order relational logic with transitive closure. We choose Alloy because of its expressive power and use in numerous domains, including security [31, 40], networking [48], and UML analysis [32, 33]. The Alloy Analyzer toolset provides an automatic analysis engine for Alloy based on off-the-shelf SAT solvers [10], which it uses to generate instances for the relations in the models such that the modeled properties either hold or are refuted, as desired.

Alloy users typically write formulas and commands to check if the model complies to a set of expected properties. For example, Pamela Zave uses a set of Alloy predicates and assertions in her model [70] to check the expected properties of the Chord [52] distributed hash table protocol. Following prior work [53], we refer to these Alloy predicates, functions and assertions that check the expected model properties as Alloy tests in the rest of this paper. These tests can help capture modeling errors and regression errors analogous to tests in imperative languages like Java. Existing debugging techniques in Alloy, e.g. MiniSat solver with unsat core [51], highlight suspicious code snippets for a single test that fails.

To improve the Alloy debugging process, we introduce AlloyFL_{hy}, the first fault localization (FL) technique that leverages multiple tests for declarative models written in Alloy. Our key insight is that a test-driven approach inspired by traditional FL based on passing and failing tests for imperative code [4, 5, 17, 20, 22, 23, 27, 28, 32, 39, 43, 45, 59, 64, 69, 71] can also lay the foundation for effective localization of faults in declarative models. Intuitively, Alloy’s expressions (including formulas) are analogous to statements in imperative languages. Alloy Expressions are hierarchical, i.e. expressions may contain other expressions, but they lack control flow.

AlloyFL_{hy} locates faults at the Abstract Syntax Tree (AST) node granularity and can locate any faulty expression in an Alloy model. An Alloy test is typically invoked with an Alloy command (i.e. run or check) with an optional "expect" constraint, where expect 1 and expect 0 indicate satisfiability and unsatisfiability of the formula being invoked, respectively. A test fails if the invocation of a command for the test is satisfiable (or unsatisfiable) but the expected result is unsatisfiable (or satisfiable). Note that any predicate, function or assertion can be treated as a test. In this paper, a fault is defined as the existence or non-existence of a set of expressions that causes some test failures. A good fault localization technique could highlight the set of expressions with the minimum number of AST nodes such that fixing those expressions make all tests pass.

We build our work on top of AUnit [53], a recent testing framework for Alloy. AUnit provides the notion of test predicates (which are ordinary Alloy predicates) that represent Alloy instances. MuAlloy [59] is a recent mutation testing framework for Alloy that can automatically generate mutant killing AUnit test predicates. Since the availability of manually written tests for real faulty Alloy models is rather limited, we use MuAlloy to generate tests to evaluate AlloyFL_{hy}. However, AlloyFL_{hy} does not require tests generated by MuAlloy. AlloyFL_{hy} can treat any predicate, function or assertion a developer provides as tests. Notably, when utilizing MuAlloy’s automatically generated tests, only a few manual steps, i.e. labeling the tests’ satisfiability, are required to create a large number of tests.

One of the backend solvers for the Alloy Analyzer is the MiniSat solver with unsat core [51, 56, 57]. The solver is
able to highlight the set of Alloy expressions for which no satisfying Alloy instance exists [1]. In this paper, we develop a more sophisticated baseline technique, i.e. AlloyFL$_{un}$, to simulate how Alloy users can debug a faulty model using the built-in MiniSat solver and a set of failing tests. AlloyFL$_{un}$ collects all Alloy AST nodes highlighted by the unsat core for each unsatisfiable failing test. Nodes highlighted more often are more likely to be faulty and are ranked at the top.

To explain AlloyFL$_{hy}$, we first introduce AlloyFL$_{co}$ and AlloyFL$_{mu}$. AlloyFL$_{co}$ implements the spectrum-based FL (SBFL) technique [4, 17, 23, 39] for Alloy. Since Alloy does not have control-flow or execution traces, all expressions in the same paragraph are either executed together or not executed at all, where an Alloy paragraph refers to any signature, predicate, function, fact or assertion. AlloyFL$_{co}$ statically analyzes Alloy paragraphs that are transitively used in each test. Then, AlloyFL$_{co}$ computes a suspiciousness score for each Alloy paragraph based on the number of passing/failing tests that invoke the paragraph and a suspiciousness formula. Finally, AlloyFL$_{co}$ ranks the paragraphs based on the suspiciousness scores in descending order. Paragraphs covered more often by the failing tests and less often by the passing tests are ranked at the top.

AlloyFL$_{mu}$ implements the mutation-based FL (MBFL) technique [38, 43] for Alloy. AlloyFL$_{mu}$ mutates Alloy AST nodes, e.g. "a && b" to "a || b", to create non-equivalent mutants and check if the test results differ compared to the original model. AlloyFL$_{mu}$ uses a suspiciousness formula to compute the suspiciousness score for each mutant based on the number of passing/failing tests that kill the mutant. A test kills a mutant if its satisfiability changes compared to that of the original model. The node whose mutation gives the highest suspiciousness score, e.g. mutations on the node make almost all failing tests pass while preserving the results of passing tests, ranks at the top.

Lastly, AlloyFL$_{hy}$ is a hybrid technique that leverages SBFL and MBFL. For each AST node, AlloyFL$_{hy}$ computes the weighted sum from the suspiciousness scores of both AlloyFL$_{co}$ and AlloyFL$_{mu}$. AlloyFL$_{co}$ is coarse-grained and cannot be more accurate than reporting an Alloy paragraph. On the other hand, AlloyFL$_{mu}$ is finer-grained but sometimes it cannot mutate any node in a paragraph, e.g. an empty paragraph in the extreme case. AlloyFL$_{hy}$ combines AlloyFL$_{co}$ and AlloyFL$_{mu}$ to enhance the accuracy.

AlloyFL$_{hy}$ does not rank all AST nodes because Alloy does not have the notion of control flow and many Alloy expressions are either executed together or not executed at all. Note that we consider an Alloy expression to be executed if that expression is translated to CNF and executed by the SAT solver. As a consequence, many expressions (AST nodes) are equally suspicious. Additionally, previous studies of FL for imperative languages have shown that users are unlikely to inspect more than a few candidates [26, 44]. Therefore, to reduce the number of highlighted AST nodes, AlloyFL$_{hy}$ only returns nodes whose suspiciousness score differs from their parent nodes.

Many existing metrics, e.g. AWE [7], EXAM [65], Ex pense [22], LIL [38] and T-score [30], may not capture the proximity between the returned AST nodes and the faulty nodes. For example, AlloyFL$_{hy}$ may return a suspicious node that is the direct parent of a faulty node but the faulty node itself does not appear in the ranked list. In this case, none of the above metrics reflect the closeness between the returned suspicious node and the faulty node. In this paper, we follow the spirit of the nearest neighbor (NN) distance metric in program dependence graphs (PDG) [7] to quantitatively measure the closeness between the ranked nodes and the faulty nodes. Specifically, we view the Alloy AST as a PDG and adapt the NN distance metric to our problem by designing three distance metrics following NN and use the existing top-k metrics [68, 72], i.e. the number of faulty nodes in the top k returned nodes, to evaluate AlloyFL$_{hy}$.

This paper makes the following contributions:

- We propose, AlloyFL$_{hy}$, the first AST node level FL technique for Alloy that leverage multiple tests.
- We follow the spirit of an existing nearest neighbor distance metric [7] and define three new distance metrics at the AST level to measure the effectiveness of AlloyFL$_{hy}$.
- We evaluate AlloyFL$_{hy}$ using 90 real faults derived from 12 existing models. The subject models all contain one or more faults. Our experimental results show that AlloyFL$_{hy}$ is substantially more effective than AlloyFL$_{un}$.
- We made the tool and the real faulty models publicly available at [https://github.com/kaiyuanw/AlloyFLCore](https://github.com/kaiyuanw/AlloyFLCore).

II. Example Model

We next present a real-world faulty Alloy model to introduce key concepts for AlloyFL$_{hy}$. We briefly describe the basics of Alloy and AUnit as needed.

Figure 1A shows a faulty Alloy model that models the Java class hierarchy constraints. The model was written by a graduate student and has two faults. Figure 1B shows two AUnit tests that fail. The tests are generated by MuAlloy [59].

The signature "(sig) declaration "(sig) Class", introduces a set of class atoms representing Java classes; "ext: lone Class" declares a field ext that relates each class to at most one class, and it represents the Java inheritance relationship. "one sig Object extends Class" introduces a special singleton Object class that represents the java.lang.Object class. The predicate ObjectNoExt should state that the Object class does not have a superclass. The predicate Acyclic should state that any class is not a subclass of itself, transitively. The predicate AllExtObject should state that every class except the Object class is a subclass of the Object class. The predicate ClassHierarchy is a conjunction of ObjectNoExt, Acyclic and AllExtObject. The run command should give an Alloy instance that represents a valid Java class hierarchy.

The student is asked to complete all predicates ObjectNoExt, Acyclic and AllExtObject. There are faults in two predicates: ObjectNoExt and AllExtObject. The predicate ObjectNoExt incorrectly
states that the Object class is not a superclass of each class, transitively. A simple fix is to replace "c.~ext" with "c.~ext" or "c.~ext", which means that the Object class is not a subclass of each class, transitively. The predicate AllExtObject incorrectly states that every class except the Object class directly extends another class that transitively extends the Object class. A simple fix is to replace "c.~ext" with either "c.~ext" or "c.~ext", which means that every class except the Object class transitively extends the Object class. There are more complicated ways to fix the predicates. In this paper, we assume that the model should be fixed by patches with a small edit distance, and we highlight code that needs to be edited accordingly with underscores in Figure 1a.

Two automatically generated AUnit tests that reveal the faults are shown in Figure 1b. Predicate test1 encodes a valuation of each signature type in Figure 1a and it represents an invalid Java class hierarchy where java.lang.Object is a subclass of another class. The Object signature contains a single atom Obj, and the Class signature contains the Obj atom and an additional Clz atom. ext relates the Obj atom to the Clz atom, and it means that the Obj class extends the Clz class. The invocation of ObjectNoExt should enforce that the Obj class does not have any superclass. Thus, test1 should be unsatisfiable (expect 0) because test1’s class hierarchy should not satisfy ObjectNoExt. Predicate test2 encodes another valuation of each signature type in Figure 1a and it represents a valid Java class hierarchy. The valuation in test2 is similar to that in test1 except that ext relates the Clz atom to the Obj atom, and it means that the Clz class extends the Object class. The invocation of AllExtObject should enforce that all classes extend the Object class. Thus, test2 should be satisfiable (expect 1) because test2’s class hierarchy should satisfy AllExtObject.

In practice, test1 is satisfiable and test2 is unsatisfiable due to the two faults in predicates ObjectNoExt and AllExtObject, resulting in test failures. We do not show all 25 tests due to space limits. Note that we use MuAlloy to generate semantically non-equivalent mutants of the model given a bound of the universe and then automatically convert the instances that differentiate each mutant from the original model into test predicate, e.g. test1. Then, we automatically label the expected satisfiability of test predicates using the correct model in our experiment. In practice, the correct model is unknown, so developers need to manually verify the satisfiability of the tests and label them accordingly.

We use a generated test suite that contains some failing tests, including test1 and test2, to locate faults in the class diagram model using both AlloyFL_un and AlloyFL_hy. AlloyFL_un serves as the baseline technique that simulates how Alloy users would debug a faulty model using the unsat core. We use the Ochiai [2] formula for AlloyFL_hy and compute a weighted sum from 60% AlloyFL_un score and 40% AlloyFL_hy score. Section V shows that this setting gives the best effectiveness. AlloyFL_un reports the entire body of AllExtObject as the most suspicious AST node (highlighted in red) and is unable to locate the fault in ObjectNoExt. AlloyFL_un only works if the failing tests are unsatisfiable which is not the case for any test reasoning over ObjectNoExt. The most suspicious nodes reported by AlloyFL_hy are highlighted in green (multiple nodes share the same highest suspiciousness score), and the second most suspicious node is highlighted in yellow. We can see that AlloyFL_hy accurately highlights the faults in both ObjectNoExt and AllExtObject. In comparison, AlloyFL_un is not able to highlight the fault in ObjectNoExt and cannot accurately highlight the fault in AllExtObject.
We first describe the formulas that compute the suspiciousness scores (Section III-A). We then discuss more details about AlloyFL\textsubscript{un}, AlloyFL\textsubscript{co}, AlloyFL\textsubscript{mu}, and finally AlloyFL\textsubscript{hy} (Section III-B).

### A. Suspiciousness Formulas

Figure 2 shows the formulas to compute suspiciousness scores, including Tarantula \cite{22}, Ochiai \cite{2}, Op2 \cite{39}, Barinel \cite{3} and DStar \cite{64}. These formulas are popular in SBFL for imperative languages. For AlloyFL\textsubscript{co}, the code elements \((e)\) are AST nodes. For AlloyFL\textsubscript{mu}, mutations of killed mutants are treated as covered code elements while mutations of live mutants are treated as uncovered code elements. \textit{totalfailed} and \textit{totalpassed} are the number of tests which failed and passed for the original model. \textit{failed(e)} and \textit{passed(e)} are the number of failing and passing tests that cover the AST node or kill the mutant \(e\). For AlloyFL\textsubscript{co}, if no passing test covers an AST node, then both Tarantula and Op2 assign a suspiciousness score of 1 to the corresponding node. The Ochiai formula assigns a suspiciousness score of 1 to a node if the node is covered by all failing tests but no passing test. Typically, if a node is covered by more failing tests but fewer passing tests, then it is assigned a higher suspiciousness score. For AlloyFL\textsubscript{mu}, if no passing test fails after mutation, Tarantula and Op2 assign a suspiciousness score of 1 to the corresponding mutated node. The Ochiai formula assigns a suspiciousness score of 1 to the mutated node if no passing test fails and all failing tests pass after the mutation. If a mutated node makes more failing tests pass but fewer passing tests fail, then it is assigned a higher suspiciousness score.

### B. AlloyFL\textsubscript{un}, AlloyFL\textsubscript{co}, AlloyFL\textsubscript{mu} and AlloyFL\textsubscript{hy}

**AlloyFL\textsubscript{un}**. We modify the standard Alloy toolset to return the AST nodes in the unsat core when the MiniSat solver is used \cite{56,57}. We configure the solver such that it is guaranteed to return a local minimum core, and all Alloy expressions are fully expanded (pushing negations in, removing existential quantifiers using skolemization and expanding universal quantifiers given the bounds on the signatures) to make the returned core as fine-grained as possible. AlloyFL\textsubscript{un} constructs a hit-map for the entire AST, and every node in the AST has a count initially set to 0. For each unsatisfiable failing test, AlloyFL\textsubscript{un} increases the count of the node and its descendants that appears in the unsat core by 1.

Figure 3 shows how the hit-map is built. Initially, each node has a count of 0 (Figure 3(a)). In Figure 3(b), a node denoted by the square is returned by the unsat core and AlloyFL\textsubscript{un} increases the counts of all affected descendants. This process applies for all the subsequently returned nodes. For example, suppose the square node in Figure 3(c) is returned next, the count of each descendant is increased to 1, and the count of each previously hit node is increased to 2. Note that a child node always has a count greater than or equal to its parent’s count. AlloyFL\textsubscript{un} collects every node whose count is strictly greater than its parent’s count, e.g. the gray nodes in Figure 3(c). The collected nodes are ranked in descending order of the corresponding count. In case of a tie, AlloyFL\textsubscript{un} prioritizes the node with a smaller number of descendants. Note that AlloyFL\textsubscript{un} only works for unsatisfiable tests and cannot be used if the model is strictly underconstrained, in which case no unsatisfiable failing test exists.

**AlloyFL\textsubscript{co}**. Since Alloy does not have control-flow and execution traces, for a given test, every code element in the same paragraph will be either executed together or not executed at all. This means nodes declared in the same paragraph share the same suspiciousness score. To implement AlloyFL\textsubscript{co}, we built a static analyzer which analyzes the entire AST and binds a variable usage a predicate/function call to its signature or predicate/function declaration, respectively. The static analyzer is able to find all Alloy paragraphs transitively used by a test, but it ignores dependencies that are never used. For example, if a test uses an expression "all s: S, t: T | some s ⋀ p[s]" where variable "t" is not used, then the test only depends on signature "S" and predicate "p[...]". By default, all facts are implicitly used, and all paragraphs transitively invoked in the facts are covered by each test. AlloyFL\textsubscript{co} computes a suspiciousness score for each Alloy paragraph based on the number of passing/failing tests that cover it and a formula shown in Figure 2. Finally, all paragraphs are ranked in descending order of suspiciousness score. In case of a tie, AlloyFL\textsubscript{co} prioritizes the paragraph with a smaller number of AST nodes.

**AlloyFL\textsubscript{mu}**. AlloyFL\textsubscript{mu} implements a wide variety of mutation operators as shown in Figure 4. MOR mutates signature...
Algorithm 1: Mutation-Based Fault Localization

Input: Faulty Alloy model $M$, test suite $T$, mutation operators $Ops$, suspiciousness formula $F$.

Output: Ranked list of suspicious AST nodes $L$. 

$L \leftarrow \emptyset$, $S \leftarrow \emptyset$, $R = \text{runTests}(M, T)$ 

$n2s \leftarrow \langle \text{ASTNode}, \text{Double}\rangle \{\}$ // Default value is 0.0

foreach $r \in R$ do

if $r.isPassed()$ then continue

foreach $n \in \text{staticAnalyze}(r)$ do

if $n \notin S$ then continue

foreach $op \in Ops$ do

if $\text{isApplicable}(op, n)$ then continue

$M' = \text{applyOp}(op, n, M)$

if $\text{isValid}(M') \& \& \text{isEquivalent}(M, M')$ then

$R' = \text{runTests}(M', T)$

score = $\text{calcSusp}(F, R, R')$

$n2s[n] = \max(n2s[n], \text{score})$

if $n2s[n] > 0.0$ then

$L.\text{add}(n)$

$L.\text{sortByScore}(n2s, \text{reverse}=\text{True})$

return $L$

---

<table>
<thead>
<tr>
<th>Mutation Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOR</td>
<td>Multiplicity Operator Replacement</td>
</tr>
<tr>
<td>QOR</td>
<td>Quantifier Operator Replacement</td>
</tr>
<tr>
<td>UOR</td>
<td>Unary Operator Replacement</td>
</tr>
<tr>
<td>BOR</td>
<td>Binary Operator Replacement</td>
</tr>
<tr>
<td>LOR</td>
<td>List Operator Replacement</td>
</tr>
<tr>
<td>UOI</td>
<td>Unary Operator Insertion</td>
</tr>
<tr>
<td>UOD</td>
<td>Unary Operator Deletion</td>
</tr>
<tr>
<td>LOD</td>
<td>Logical Operand Deletion</td>
</tr>
<tr>
<td>PBD</td>
<td>Paragraph Body Deletion</td>
</tr>
<tr>
<td>BOE</td>
<td>Binary Operator Exchange</td>
</tr>
<tr>
<td>IEOE</td>
<td>Imply-Else Operand Exchange</td>
</tr>
</tbody>
</table>

Fig. 4: Mutation Operators.

Multiplicities, e.g. "one sig" to "lone sig". QOR mutates quantifiers, e.g. some to all. UOR, BOR and LOR define operator replacement for unary, binary and list operators, respectively. For example, UOR mutates $a\cdot b$ to $a\cdot b$; BOR mutates $a<=>b$ to $a>b$; and LOR mutates $a|b$ to $a\&b$. UOI inserts an unary operator before expressions, e.g. $a\cdot b$ to $a\cdot b$. UOD deletes an unary operator, e.g. $a\cdot b$ to $a\cdot b$. LOD deletes an operand of a logical operator, e.g. $a\&b$ to $a\&b$. PBD deletes the body of a paragraph. BOE exchanges operands for a binary operator, e.g. $a\cdot b$ to $b\cdot a$. IEOE exchanges the operands of imply-else operation, e.g. "a => b else c" to "a => c else b".

Algorithm 1 shows the details of AlloyFL$_{mu}$. The algorithm takes as input a faulty Alloy model $M$, a test suite $T$, a set of mutation operators $Ops$ and a suspiciousness formula $F$. The output of the algorithm is a ranked list of suspicious AST nodes ($L$) sorted in the descending order of suspiciousness. Initially, $L$ is set to an empty list. $S$ keeps the set of nodes covered by failing tests and is initialized as an empty set. AlloyFL$_{mu}$ runs $T$ against $M$ and stores the results in $R$. $n2s$ keeps the mapping from a node to its suspiciousness score and is initialized to an empty map.

For each test result $r$ in $R$, AlloyFL$_{mu}$ collects nodes and their descendants covered by all failing tests. Then, AlloyFL$_{mu}$ iterates over each node $n$ in $M$. If $n$ is not covered by any failing test, i.e. $n \notin S$, then AlloyFL$_{mu}$ skips it. For each $n$ covered by a failing test, AlloyFL$_{mu}$ tries to apply every mutation operator in $Ops$ to the node, one at a time. If the mutation operator is not applicable, it is skipped. Otherwise, AlloyFL$_{mu}$ mutates $M$ to $M'$. If $M'$ leads to a compilation error or is equivalent to $M$, then AlloyFL$_{mu}$ skips $M'$. Otherwise, AlloyFL$_{mu}$ runs $T$ against the mutant $M'$ and collects the result as $R'$. Function calcSusp computes the suspiciousness score of the mutant based on the formula $F$ (Figure 2), and test results $R$ and $R'$. $n2s$ keeps the maximum suspiciousness score for each node $n$. After AlloyFL$_{mu}$ exhausts all mutation operators that are applicable to $n$, $n$ is added to $L$ if its suspiciousness score $n2s[n]$ is greater than 0. Finally, after all AST nodes are exhausted, $L$ is sorted in descending order of suspiciousness scores. Note that we check the equivalence between the mutated model and the original model by constructing an Alloy assertion that checks if the mutated paragraph is equivalent to the original paragraph given a bound of the declared signatures [54].

AlloyFL$_{hy}$. AlloyFL$_{hy}$ is a hybrid technique that leverages both AlloyFL$_{co}$ and AlloyFL$_{mu}$. Given an AST node, AlloyFL$_{co}$ computes a score $S_{co}$ and AlloyFL$_{mu}$ computes a score $S_{mu}$. AlloyFL$_{hy}$ computes the weighted sum as $(1-\lambda)S_{co} + \lambda S_{mu}$, where $0 \leq \lambda \leq 1$. If no mutation applies to a node, AlloyFL$_{hy}$ uses $S_{co}$. The intuition is that AlloyFL$_{mu}$ sometimes performs badly for omission errors in which case AlloyFL$_{co}$ performs relatively well. Thus, AlloyFL$_{hy}$ benefits from both AlloyFL$_{co}$ and AlloyFL$_{mu}$.

IV. Distance Metrics

To quantitatively measure how close the ranked nodes are to the real faulty nodes, we follow the spirit of the nearest neighbor distance metric (NN) based on program dependence graphs (PDG) [47]. Since Alloy does not have control dependencies, we view the Alloy AST as a PDG and adapt the NN distance metric to reason over the AST.

The original nearest neighbor distance metric quantifies the percentage of nodes not needing inspection by the programmer using the formula $1 - \frac{\sum_{i=1}^{k} \text{nodes}}{|R|}$, where $R = \{n_1, n_2, ..., n_k\}$ are the top $k$ returned suspicious nodes, and $S(R)$ is a sphere of all nodes in the graph $G$ such that the maximum distance of any node in $S$ to its closest suspicious node is smaller or equal to the minimum distance of any suspicious node in $R$ to its closest faulty node. Conceptually, the user does a breadth-first search starting with the suspicious nodes, and increasing the distance until a defect is found. The formula computes the percentage of nodes that need not be examined. However,
previous studies show that: (1) the percentage of nodes needing inspection is a better estimate than the percentage of nodes not needing inspection [30, 65]; and (2) fault localization techniques should focus on improving absolute rank rather than percentage rank [44]. Thus, we adapt the NN metric to use the absolute number of nodes needing inspection (|S(R)|). Techniques which give smaller distance metric values are more accurate. We next describe three distance metrics.

**Nearest Neighbor Up-Down (NNUD).** NNUD sets R to the k most suspicious nodes returned. It allows traversing upward (parent) and downward (children) from the suspicious nodes in the AST until a faulty node is found. In other words, NNUD assumes that the programmer may look at the parent or children when inspecting the top k suspicious nodes until a faulty node is found. Figure 5(a) shows the number of nodes one needs to explore from the top two suspicious nodes. The number in the circle represents the position of the node in the ranked list, e.g. 1 means it ranks at the top. "F" shows the faulty node and squares are irrelevant nodes. Circles colored in gray estimate the nodes users need to inspect under NNUD metric with k = 2. Since the minimum distance between any of the two suspicious nodes and the faulty node is 1, all nodes that are reachable from the suspicious nodes within a distance of 1 are included. Thus, the metric reports 6, i.e. the number of the gray nodes.

**Nearest Neighbor Down (NND).** NND does not allow traversing upward from the suspicious node and it processes suspicious nodes one at a time. This metric assumes that the user only inspects the children and will never inspect already visited nodes. Figure 5(b) shows how the metric works. From the top most suspicious node, we can only traverse downward. Since no faulty node is found, we mark all inspected nodes in gray. Then, NND does a breadth-first search for the second top suspicious node. In this case, a faulty node is found and all descendants within the same distance, i.e. 1, are included (3 circles colored in white), excluding already visited nodes colored in gray. Finally, NND reports 6, i.e. the number of the inspected nodes in circles. However, it is possible that the faulty node never appear as the descendants of any suspicious node. To avoid this scenario, we append the root node of the entire AST to the end of the suspicious node list.

**Nearest Neighbor Down Worst (NNDW).** NNDW is similar to NND (only allows traversing downward) except that it assumes the user is unlucky and would inspect all non-faulty nodes before finding the fault. Figure 5(c) shows how the metric works. Inspecting the top suspicious node is similar to NND, with the difference occurring when inspecting the second top suspicious node. In this case, we traverse downward and include all non-faulty nodes that have not been visited before (white circles without the faulty node). If a faulty node can be reached from the current suspicious node, then we stop traversing and include all such faulty nodes. In this case, two faulty nodes appear as the children of the second top suspicious node, so we include both faulty nodes. Finally, NNDW returns 10, i.e. all circle nodes. Similar to NND, we append the root node of the entire AST to the end of the suspicious node list.

## V. Evaluation

We evaluate AlloyFL-hy on 90 real faults collected from Alloy release 4.1, Amalgam [41] and graduate student solutions. These faulty models contain various types of faults, including overconstraints, underconstraints and a mixture of both. All experiments are performed on Linux 5.2.17 with 2.2GHz Intel Xeon CPU and 32 GB memory.

In this section, we address the following research questions:

**RQ1.** How does the suspiciousness formula affect AlloyFL-hy?

**RQ2.** How does the AlloyFL-mut weight λ affect AlloyFL-hy?

**RQ3.** What is the effectiveness of AlloyFL-uns, and AlloyFL-hy?

**RQ4.** How does the test size affect AlloyFL-hy?

**RQ5.** What is the time overhead of AlloyFL-hy?

### A. Experiment Setting

Figure 6 gives an overview for the 12 correct models used to generate mutant faults in the evaluation. Address book (addr) and farmer cross-river puzzle (farmer) are from Alloy’s example set. Bad employee (bempl), grade book (grade), and other groups (other) are Alloy translations of access-control specifications used to benchmark Amalgam [41]. Colored tree (ctree) is from MuAlloy [55]. Array model (array), balanced binary search tree (bst), class diagram (cd), doubly-linked list (dll), finite state machine (fsm), and singly-linked list with sorting and counting functions (scl) are homework questions we collected from graduate students.

For each subject, Figure 6 shows the number of AST nodes (ast), the number of nonequivalent first-order mutants (mut), the number of tests automatically generated (tot), the number of tests that are expected to be satisfiable (sat) and unsatisfiable (uns), and the scope used to run tests or equivalence
check results. For AlloyFL$_{hy}$, Ochiai gives the best result for all metrics except NND, NNDW, top-5 and top-10 metrics, where these metrics are only slightly worse compared to the best formulas. Op2 gives the worst result and DStar gives the second worst result. Tarantula and Barinel are comparable and slightly worse than Ochiai. Overall, the choice of formulas (except Op2) does not impact the accuracy of AlloyFL$_{co}$ much, and the Ochiai formula is the best choice for both AlloyFL$_{mu}$ and AlloyFL$_{hy}$. We use the Ochiai formula in the rest of the evaluation.

C. RQ2: AlloyFL$_{mu}$ Weight Impact

Figure 8 shows the average distance metrics and the sum of top-k metrics for different AlloyFL$_{mu}$ weight $\lambda$ in AlloyFL$_{hy}$. The best weights are highlighted in bold and blue for each metric. When $\lambda = 0.0$, AlloyFL$_{hy}$ is equivalent to AlloyFL$_{co}$. When $\lambda = 1.0$, AlloyFL$_{hy}$ is equivalent to AlloyFL$_{mu}$. The results show that AlloyFL$_{hy}$ achieves the best performance when $\lambda = 0.4$.

D. RQ3: AlloyFL$_{hy}$ Effectiveness

Figure 9 shows the distance and top-k metrics results for AlloyFL$_{un}$, AlloyFL$_{co}$, AlloyFL$_{mu}$ and AlloyFL$_{hy}$. We use the Ochiai formula for AlloyFL$_{co}$, AlloyFL$_{mu}$ and AlloyFL$_{hy}$. The AlloyFL$_{mu}$ weight is set to 0.4 for AlloyFL$_{hy}$. The most accurate techniques are highlighted in bold and blue for each fault and metric. The #Fault column shows the number of actual faults in each model. The bottom of Figure 9 gives a summary of the performance of each technique across all faults. Avg/Sum shows the average of distance metrics per technique, the sum of top-k metrics per technique, and the total number of faults across all faulty models. Win shows the number of times the corresponding technique gives the best result for each metric. Un, Co, Mu and Hy represent AlloyFL$_{un}$, AlloyFL$_{co}$, AlloyFL$_{mu}$ and AlloyFL$_{hy}$, respectively. Note that the sorting method might be unstable, so the final result might vary slightly for different runs, e.g. metric values when $\lambda = 0$ or 1 in Figure 8 is slightly different from the values in Figure 8.

With AlloyFL$_{hy}$, users can find at least one fault by inspecting 9.2, 10.4 and 14.1 AST nodes from the top 1, 5 and 10 reported nodes under NNUD, respectively. Users need to inspect 6.0 and 8.2 AST nodes to find at least one

Fig. 8: AlloyFL$_{mu}$ Weight Impact for AlloyFL$_{hy}$.
Fig. 9: Effectiveness for Real Faults.
Fig. 10: Test Size Impact for AlloyFL.<n>AlloyFL<sub>co</sub> is accurate for omission faults, e.g. when users leave the entire predicate body empty (scl16) or when the user misses some conjunct/disjunct constraints at the body level of a predicate (grade1). On the contrary, AlloyFL<sub>mu</sub> is more accurate for faults that can be fixed with mutations, e.g. addr1 and bst3. AlloyFL<sub>hy</sub> performs, on average, better than AlloyFL<sub>co</sub> and AlloyFL<sub>mu</sub> because it benefits from the strengths of both AlloyFL<sub>co</sub> and AlloyFL<sub>mu</sub>. On the other hand, AlloyFL<sub>un</sub> prioritizes AST nodes that are highlighted the most number of times by the unsat core across all unsatisfiable failing tests, so it is comparable or more accurate than using a single unsatisfiable failing test, i.e. the traditional way an Alloy user would debug a faulty model using the unsat core. However, since the unsat core is still too coarse grained and cannot handle underconstrained models, AlloyFL<sub>un</sub> cannot locate faults accurately. Finally, our results also confirm our hypothesis and show that AlloyFL<sub>hy</sub> is substantially more effective than AlloyFL<sub>un</sub> under all metrics.

E. RQ4: Test Size Impact

Figure 10 shows the average distance metrics and the sum of top-k metrics for different test sizes for AlloyFL<sub>hy</sub> (with Ochiai formula and AlloyFL<sub>mu</sub> weight λ = 0.4). The ratio column shows the test size in percentage we randomly selected from the full test suite. For each test size, we run the experiment for 10 trials and report the average results. For each trial, we make sure that any test suite with a smaller size is a subset of any test suite with a larger size. AlloyFL<sub>hy</sub> reaches the best performance under all metrics when using the full test suite. The effectiveness of AlloyFL<sub>hy</sub> decreases as the test size decreases.

Specifically, AlloyFL<sub>hy</sub> is inaccurate with only 10% of the tests. It is much more accurate with >50% of the tests. The average effectiveness increases slowly as the test ratio decreases from 50% to 100%. Moreover, we observe that some specific trials with a test ratio <100% can give better results than using the full test suite. This is due to the randomness as sometimes computing the suspiciousness scores from the sampled tests gives more accurate rankings. To make the result reliable and stable, AlloyFL<sub>hy</sub> should use the entire test suite.

F. RQ5: AlloyFL Time Overhead

Figure 11 shows the time overhead in seconds for each model. Avg shows the average time overhead over all faulty models for each technique. We use the Ochiai formula.

Fig. 11: Time Overhead (sec) for Real Faults.
for AlloyFL$_{co}$, AlloyFL$_{mu}$ and AlloyFL$_{hy}$. The AlloyFL$_{mu}$ weight is set to 0.4 for AlloyFL$_{hy}$. We observe that AlloyFL$_{un}$ and AlloyFL$_{co}$ are comparable and it takes both techniques less than 6 seconds to finish for each model. AlloyFL$_{mu}$ and AlloyFL$_{hy}$ are comparable and they are slower than both AlloyFL$_{un}$ and AlloyFL$_{co}$ for each model. AlloyFL$_{mu}$ and AlloyFL$_{hy}$ take a minimum of 2.0 and 2.1 seconds, respectively, to finish for cd2. They take a maximum of 134.3 and 136.4 seconds, respectively, to finish for dll18. A majority of AlloyFL$_{hy}$’s time overhead is attributed to AlloyFL$_{un}$. On average, AlloyFL$_{un}$, AlloyFL$_{co}$, AlloyFL$_{mu}$, and AlloyFL$_{hy}$ finish in 2.1, 2.1, 30.5 and 30.7 seconds, respectively. Since AlloyFL$_{hy}$ finishes around 30 seconds on average, its time overhead is acceptable because it is substantially more accurate than AlloyFL$_{un}$.

G. Threats to Validity

There exists several threats to the validity of our results. The real faulty models we use in the experiment are limited in the sense that most of them are written by graduate students (with few real faults written by experienced developers). Therefore, the results may not generalize to faulty models written by experienced developers. However, we collected faulty models to the best of our ability.

The best AlloyFL$_{mu}$ weight 0.4 is chosen based on the experimental faulty models, so it may not generalize to unseen faulty models.

The tests used to capture desired model properties (e.g. the test in Figure 1b) can require some effort to create. In this paper, all tests are automatically generated using MuAlloy [59] and the expected behavior (expect 0 or expect 1) of each test is automatically verified using the correct model. In practice, users need to specify the expected behavior but no manual effort is needed to create test predicates if users choose to use MuAlloy. We use generated test predicates to evaluate AlloyFL since we did not find a reasonably large set of manually written tests for every real faulty model. So our results may not generalize to manually written tests.

Additionally, although our distance metrics simulate different ways users may inspect code highlighted by AlloyFL, users may use the reported Alloy expressions in a different manner. Nevertheless, we also evaluate AlloyFL$_{hy}$ using the traditional top-k metrics to enhance the credibility of the conclusion.

VI. RELATED WORK

This paper presents AlloyFL$_{hy}$ – the first automated fault localization technique for Alloy that leverages multiple test predicates. AlloyFL$_{hy}$ highlights suspicious code more accurately than the unsat cores based technique AlloyFL$_{un}$. Moreover, AlloyFL$_{hy}$ enables the evaluation of program repair techniques, e.g. ARepair [58] for Alloy models. Note that ARepair assumes that the faulty locations are given and it focuses on synthesizing code snippets. On the contrary, AlloyFL$_{hy}$ focuses on locating faults.

Automated debugging of Alloy models can be traced back to Alloy’s early days when highlighting unsat cores in unsatisfiable Alloy expressions is introduced [51]. Moreover, for satisfiable expressions, Alloy’s symmetry breaking indirectly supports debugging by allowing the user to inspect fewer instances [13, 42, 50]. More recent work on Amalgam allows the user to ask questions of the form “why a tuple is or is not in a relation” for a chosen instance [41]. While Amalgam provides a useful tool to aid debugging by allowing the user to enhance their understanding of the model, the restricted form of the questions users can ask limits its effectiveness, e.g. the user cannot ask why certain expressions hold or not, or why certain relations are empty.

A number of approaches assist users in writing correct Alloy models. Montaghami and Rayside [36, 37] enable Alloy users to more easily provide partial instances, which are expressive example solutions that aid in writing correct, complete models. Our prior work [55] follows the spirit of JUnit and introduces a test automation framework for Alloy by defining test, test execution and model coverage. AUnit has further enabled test automation efforts for Alloy, ranging from automated test generation to mutation testing [54, 59]. ASketch helps users to generate complicated Alloy expressions based on a partial Alloy model and a set of tests [60, 61]. Other techniques have been developed to run a subset of AUnit tests [62] or reduce the test execution time [63].

While our focus in this paper is on declarative models written in Alloy, fault localization for imperative languages is a well-studied area. AlloyFL$_{co}$, AlloyFL$_{mu}$ and AlloyFL$_{un}$ implement spectrum-based, mutation-based, and SAT-based techniques, respectively. Among these, spectrum-based techniques [2, 4, 6, 8, 11, 12, 16, 22, 23, 25, 29, 33, 46, 47, 60, 67, 73], are the most widely studied; they focus on collecting execution information, such as statements and methods. Mutation-based fault localization techniques [15, 38, 43] were introduced more recently. They perform mutations on the faulty program to study their impact on the test results and determine likely faulty locations. SAT-based techniques use either the minimal satisfiability [14] or the negation of maximal satisfiability [24] to identify suspicious code.

VII. CONCLUSIONS

This paper introduces AlloyFL$_{hy}$, a fault localization technique for declarative Alloy models. AlloyFL$_{hy}$ is the first technique that utilizes a suite of test predicates (either automatically generated or manually written) that capture the desired properties of Alloy models to locate faults at the AST node granularity. We also propose new distance metrics, i.e. NNUD, NND and NNDW, to evaluate AlloyFL$_{hy}$. The evaluation is performed on 90 real faulty models and shows that AlloyFL$_{hy}$ is substantially more effective than the baseline technique AlloyFL$_{un}$. We also show that using the Ochiai formula and setting the AlloyFL$_{mu}$ weight to 0.4 make AlloyFL$_{hy}$ achieve the best effectiveness.

ACKNOWLEDGMENTS

This work was partially supported by NSF grant nos. CNS-1646305, CCF-1718903, CCF-1918189 and CCF-1956374.
REFERENCES


[43] M. Papadakis and Y. Le Traon. Metallaxis-fl: Mutation-
based fault localization. STVR, 2015.