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 real faulty models. The results show that AlloyFL$_{hy}$ is substantially more effective than Alloy's built-in unsat core.**

***Keywords*-Alloy, Fault localization, AlloyFLHy**

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, 38, 39, 43, 45, 49, 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], Expense [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) [47] 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 [47] 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.

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

```
sig Class {
  ext: lone Class }

one sig Object extends Class {}

pred ObjectNoExt() {
  // Object does not extend any class.
  // Fix: replace c.^ext with c.~^ext or c.^~ext.
  all c: Class | Object !in c.^ext }

pred Acyclic() {
  // Each class is not a subclass of itself.
  all c: Class | c !in c.^ext }

pred AllExtObject() {
  // Each class is a subclass of the Object class.
  // Fix: replace c.ext.^ext with c.^ext or
  //      replace c.ext.^ext with c.ext.*ext.
  all c: Class - Object | Object in c.ext.^ext }

pred ClassHierarchy() {
  ObjectNoExt
  Acyclic
  AllExtObject }

run ClassHierarchy for 3
```

(a) Faulty Java class diagram model

```
// class Clz {}
// class Object extends Clz {}
pred test1 {
  some disj Obj: Object |
  some disj Obj, Clz: Class {
    Object = Obj
    Class = Obj + Clz
    ext = Obj->Clz
    ObjectNoExt[]
  }
}
run test1 for 3 expect 0 // Not allowed

// class Object {}
// class Clz extends Object {}
pred test2 {
  some disj Obj: Object |
  some disj Obj, Clz: Class {
    Object = Obj
    Class = Obj + Clz
    ext = Clz->Obj
    AllExtObject[]
  }
}
run test2 for 3 expect 1 // Allowed
// More tests ...
```

(b) MuAlloy generated tests

Fig. 1: Faulty Java class diagram and generated tests

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.^ext`" with either "`c.^ext`" or "`c.ext.*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$_{co}$ score and 40% AlloyFL$_{mu}$ 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`.

Name	Formula
Tarantula [22]	$\frac{\frac{failed(e)}{totalfailed}}{\frac{failed(e)}{totalfailed}+\frac{passed(e)}{totalpassed}}$
Ochiai [2]	$\frac{failed(e)}{\sqrt{totalfailed \times (failed(e)+passed(e))}}$
Op2 [39]	$failed(e) - \frac{passed(e)}{totalpassed+1}$
Barinel [3]	$1 - \frac{passed(e)}{passed(e)+failed(e)}$
DStar [64]	$\frac{failed(e)^{*}}{passed(e)+(totalfailed-failed(e))}$
totalfailed: total number of tests that failed *totalpassed*: total number of tests that passed *failed(e)*: number of failed tests that cover or kill *e* *passed(e)*: number of passed tests that cover or kill *e*	

Fig. 2: Suspiciousness Formulas

III. Technique

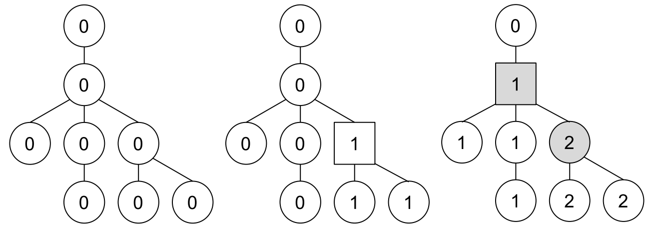
We first describe the formulas that compute the suspiciousness scores (Section III-A). We then discuss more details about AlloyFL$_{un}$, AlloyFL$_{co}$, AlloyFL$_{mu}$ and finally AlloyFL$_{hy}$ (Section III-B).

A. Suspiciousness Formulas

Figure 2 shows the formulas to compute suspiciousness scores, including Tarantula [22], Ochiai [2], Op2 [39], Barinel [3] and DStar [64]. These formulas are popular in SBFL for imperative languages. For AlloyFL$_{co}$, the code elements (*e*) are AST nodes. For AlloyFL$_{mu}$, mutations of killed mutants are treated as covered code elements while mutations of live mutants are treated as uncovered code elements. *totalfailed* and *totalpassed* are the number of tests which failed and passed for the original model. *failed(e)* and *passed(e)* are the number of failing and passing tests that cover the AST node or kill the mutant *e*. For AlloyFL$_{co}$, 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$_{mu}$, 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$_{un}$, AlloyFL$_{co}$, AlloyFL$_{mu}$ and AlloyFL$_{hy}$

AlloyFL$_{un}$. We modify the standard Alloy toolset to return the AST nodes in the unsat core when the MiniSat solver is used [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$_{un}$ 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$_{un}$ increases the count of the node and its descendants that appears in the unsat core by 1.



(a) No hit (b) 1st hit (c) 2nd hit

Fig. 3: Illustration of AlloyFL$_{un}$

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$_{un}$ 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$_{un}$ 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$_{un}$ prioritizes the node with a smaller number of descendants. Note that AlloyFL$_{un}$ only works for unsatisfiable tests and cannot be used if the model is strictly underconstrained, in which case no unsatisfiable failing test exists.

AlloyFL$_{co}$. 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$_{co}$, 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$_{co}$ 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$_{co}$ prioritizes the paragraph with a smaller number of AST nodes.

AlloyFL$_{mu}$. AlloyFL$_{mu}$ implements a wide variety of mutation operators as shown in Figure 4. *MOR* mutates signature

Mutation Operator	Description
MOR	Multiplicity Operator Replacement
QOR	Quantifier Operator Replacement
UOR	Unary Operator Replacement
BOR	Binary Operator Replacement
LOR	List Operator Replacement
UOI	Unary Operator Insertion
UOD	Unary Operator Deletion
LOD	Logical Operand Deletion
PBD	Paragraph Body Deletion
BOE	Binary Operand Exchange
IEOE	Imply-Else Operand Exchange

Fig. 4: Mutation Operators.

multiplicity, 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.^b` to `a.*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.b` to `a.~b`. *UOD* deletes an unary operator, e.g. `a.^~b` to `a.^b`. *LOD* deletes an operand of a logical operator, e.g. `a&&b` to `b`. *PBD* deletes the body of a paragraph. *BOE* exchanges operands for a binary operator, e.g. `a-b` to `b-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

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 ← [], S ← ∅, R = runTests(M, T)
n2s ← <ASTNode, Double>{} // Default value is 0.0
foreach r ∈ R do
    if r.isPassed() then continue
    foreach n ∈ staticAnalyze(r) do
        S = S ∪ n.getDescendants()

foreach n ∈ M.getNodes() do
    if n ∉ S then continue
    foreach op ∈ Ops do
        if !isApplicable(op, n) then continue
        M' = applyOp(op, n, M)
        if isValid(M') && !isEquivalent(M, M') then
            R' = runTests(M', T)
            score = calcSusp(F, R, R')
            n2s[n] = max(n2s[n], score)

    if n2s[n] > 0.0 then
        L.add(n)

L.sortByScore(n2s, reverse=True)
return L
```

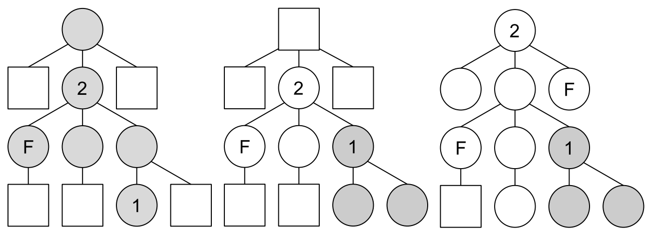
the original model by constructing an Alloy `assertion` that checks if the mutated paragraph is equivalent to the original paragraph given a bound of the declared signatures [54].

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 $1 - \frac{|S(R)|}{|G|}$, 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,



(a) NNUD $k = 2$ (b) NND (c) NNDW

Fig. 5: Distance Metrics Examples

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 reinspect 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. $\leqslant 1$, 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 nodes 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 ranked 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$_{mu}$ weight λ affect AlloyFL$_{hy}$?
RQ3. What is the effectiveness of AlloyFL$_{un}$ 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 [59]. 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.

Model	**ast**	**mut**	**test**			**scp**
			tot	**sat**	**uns**	
addr	124	62	30	19	11	3
array	68	51	38	15	23	3
bst	175	167	110	50	60	4
bempl	57	35	25	11	14	3
cd	52	46	25	9	16	3
ctree	76	83	22	9	13	3
dll	92	81	49	23	26	3
farmer	180	106	56	33	23	4
fsm	85	63	15	3	12	3
grade	77	44	41	23	18	3
other	40	32	21	9	12	3
scl	201	143	87	40	47	3
Sum	1.2k	913	519	244	275	

Fig. 6: Correct Models Information.

For each subject, Figure 6 shows the number of AST nodes (**ast**), the number of nonequivalent first-order mutants [21] (**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

	Formula	nnud-1	nnud-5	nnud-10	nnd	nndw	top-1	top-5	top-10
Co	Tarantula	**29.0**	**39.9**	**45.6**	**15.8**	**22.2**	**4**	**10**	**13**
	Ochiai	29.2	**39.9**	**45.6**	17.0	24.1	**4**	6	12
	Op2	35.9	42.0	**45.6**	26.0	34.4	2	4	12
	Barinel	**29.0**	**39.9**	**45.6**	**15.8**	**22.2**	**4**	**10**	**13**
	DStar	30.0	40.8	**45.6**	18.8	26.2	3	6	12
Mu	Tarantula	13.6	13.0	15.7	**8.7**	**13.1**	28	61	76
	Ochiai	**11.7**	12.1	**14.2**	**8.7**	**13.1**	**29**	**68**	**81**
	Op2	13.7	12.3	14.8	10.1	15.2	22	67	80
	Barinel	13.6	13.0	15.7	**8.7**	**13.1**	28	61	76
	DStar	11.9	**11.7**	14.3	9.0	13.8	23	67	80
Hy	Tarantula	9.9	11.1	14.3	5.9	8.0	25	64	**85**
	Ochiai	**9.2**	**10.4**	**14.1**	6.0	8.2	**32**	65	77
	Op2	24.6	20.0	18.0	18.6	22.7	16	44	67
	Barinel	9.7	10.6	14.3	**5.8**	**7.9**	27	**66**	**85**
	DStar	12.3	**10.4**	14.3	7.4	10.0	25	64	77

Fig. 7: Suspiciousness Formula Impact on AlloyFL$_{co}$, AlloyFL$_{mu}$ and AlloyFL$_{hy}$.

λ	nnud-1	nnud-5	nnud-10	nnd	nndw	top-1	top-5	top-10
0.0	29.2	39.9	45.6	16.7	23.8	4	6	12
0.1	9.4	**10.4**	**14.1**	6.1	8.5	31	65	78
0.2	9.4	**10.4**	**14.1**	6.1	8.5	30	65	78
0.3	9.3	**10.4**	**14.1**	6.1	8.3	**32**	63	76
0.4	**9.2**	**10.4**	**14.1**	**6.0**	**8.2**	**32**	65	77
0.5	**9.2**	10.8	14.3	6.1	8.3	31	65	77
0.6	**9.2**	10.8	14.4	6.1	8.3	31	65	78
0.7	**9.2**	10.8	14.4	6.2	8.3	30	66	78
0.8	**9.2**	10.9	14.4	6.2	8.3	30	66	78
0.9	**9.2**	11.0	14.4	6.2	8.3	30	65	78
1.0	11.7	11.8	14.2	8.7	13.1	29	**68**	**82**

Fig. 8: AlloyFL$_{mu}$ Weight Impact for AlloyFL$_{hy}$.

checks (**scp**). Prior work shows that tests generated by MuAlloy are effective in detecting real faults [54, 59]. Therefore, to produce the test suite, we use MuAlloy to generate non-equivalent mutants. For each non-equivalent mutant, MuAlloy creates an Alloy instance that differentiates the original model and the mutant, and then converts that instance into a test predicate. The expected satisfiability (whether `expect 0` or `expect 1`) of these test predicates is automatically verified by the correct model. In practice, the test satisfiability should be verified by developers.

We manually inspect all of the faults and try to fix them without changing the model structure. For example, if the model has a fault in the quantifier body, then we try to fix it without replacing the entire quantifier expression. The expressions modified due to the fix are labeled as faulty. We collect 5 real faults from [41], 1 real fault from Alloy release 4.1 and 84 real faults from graduate students.

To evaluate AlloyFL$_{hy}$, we use distance metrics (NNUD-k, NND, NNDW) and traditional top-k metrics. The distance metrics measure the number of nodes to inspect before finding at least one fault. The top-k metrics measure the number of faults found by inspecting the top-k nodes. We pick k up to 10 because [26] showed that 98% of practitioners consider a fault localization technique to be useful only if the fault appears in the top-10 suspicious elements. Techniques that give smaller distance metrics and larger top-k metrics are more accurate. Intuitively, the NNUD-1, NND, NNDW and the top-1 metrics are more important than the NNUD-5, NNUD-10, top-5 and top-10 metrics. Because users are likely to debug one fault at a time instead of looking at all faults at once.

B. RQ1: Suspiciousness Formula Impact

Figure 7 shows the average distance metrics and the sum of top-k metrics for different suspiciousness formulas for AlloyFL$_{co}$, AlloyFL$_{mu}$ and AlloyFL$_{hy}$ (with AlloyFL$_{mu}$ weight $\lambda = 0.4$). The best formulas are highlighted in bold and blue for each technique and metric. For AlloyFL$_{co}$, Tarantula and Barinel give the best result while Op2 gives the worst result . Ochiai and DStar are similar and slightly worse than Tarantula and Barinel. For AlloyFL$_{mu}$, Ochiai gives the best result for all metrics except NNUD-5. Other formulas give slightly worse 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 λ 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 **#Flt** 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 9.

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

Model	nnud-1				nnud-5				nnud-10				nnd				nndw				top-1				top-5				top-10				#Flt
	Un	Co	Mu	Hy	Un	Co	Mu	Hy	Un	Co	Mu	Hy	Un	Co	Mu	Hy	Un	Co	Mu	Hy	Un	Co	Mu	Hy	Un	Co	Mu	Hy	Un	Co	Mu	Hy	
addr1	24	78	1	1	57	78	5	5	57	73	10	10	10	50	1	1	15	57	1	1	0	0	1	1	0	0	1	1	0	0	1	1	1
arr1	20	51	14	14	56	51	38	34	61	51	38	49	61	32	8	8	64	41	15	15	0	0	0	0	0	0	0	0	0	0	0	0	1
arr2	1	29	1	1	2	26	4	5	2	26	4	8	1	22	1	1	3	52	3	3	1	0	1	1	1	0	1	1	1	0	1	1	2
arr3	1	3	3	3	3	14	3	5	3	17	3	7	1	2	2	2	3	4	4	4	1	0	0	0	1	0	1	1	1	0	1	1	1
arr4	1	62	1	1	3	62	5	5	3	62	6	10	1	32	1	1	1	44	1	1	1	0	1	1	1	0	1	1	1	0	1	1	1
arr5	1	60	1	1	3	60	5	5	3	60	6	10	1	38	1	1	1	53	1	1	1	0	1	1	1	0	1	1	1	0	1	1	1
arr6	45	45	45	45	45	40	18	13	45	40	18	22	45	26	4	4	68	38	13	13	0	0	0	0	0	0	0	0	0	0	0	0	1
arr7	1	19	1	1	3	14	5	5	3	17	6	9	1	19	1	1	1	22	1	1	1	0	1	1	2	0	2	1	2	0	2	2	2
arr8	1	3	1	1	2	14	5	5	2	17	6	10	1	2	1	1	3	5	1	1	1	0	1	1	1	0	1	1	1	0	1	1	1
arr9	53	53	21	21	53	44	5	5	53	44	7	10	53	31	4	4	84	44	5	5	0	0	0	0	0	0	1	1	0	0	1	1	1
arr10	57	48	57	57	43	48	57	57	52	48	57	57	31	50	57	50	50	72	78	72	0	0	0	0	0	0	0	0	0	0	0	0	1
arr11	75	75	1	1	75	64	5	5	75	64	7	10	75	42	1	1	98	53	1	1	0	0	1	1	0	0	1	1	0	0	1	1	1
bst1	32	32	5	5	55	58	5	5	55	65	6	10	5	5	5	5	5	5	5	5	0	0	0	0	0	0	1	1	0	0	1	1	1
bst2	3	3	4	1	15	15	14	5	15	28	10	10	2	2	7	1	6	6	7	5	0	0	0	1	0	0	0	1	0	0	1	1	4
bst3	68	19	1	1	34	47	5	5	34	62	10	10	14	6	1	1	17	9	1	1	0	0	1	1	0	0	2	1	0	0	3	1	3
bst4	3	3	64	3	15	5	53	5	27	10	59	10	2	2	50	2	2	2	54	2	0	0	0	0	0	1	0	1	0	1	0	1	3
bst5	32	32	5	5	57	59	5	5	57	67	6	10	5	5	5	5	5	5	5	5	0	0	0	0	0	0	1	1	0	0	1	1	1
bst6	108	69	1	1	78	81	5	5	78	100	8	10	34	19	1	1	36	21	1	1	0	0	1	1	0	0	1	1	0	0	1	1	1
bst7	58	58	4	4	80	82	5	5	80	89	6	10	6	6	4	4	6	6	4	4	0	0	0	0	0	0	1	1	0	0	1	1	1
bst8	57	57	4	4	79	82	5	5	79	97	10	10	6	6	4	4	6	6	4	4	0	0	0	0	0	0	1	1	0	0	1	1	3
bst9	58	58	4	4	84	90	5	5	84	99	6	10	6	6	4	4	6	6	4	4	0	0	0	0	0	0	1	1	0	0	1	1	1
bst10	32	32	5	32	33	56	5	12	45	58	10	10	5	5	5	5	5	5	5	5	0	0	0	0	0	0	1	0	0	0	3	1	4
bst11	67	34	69	69	15	15	69	69	34	8	64	64	23	51	129	51	23	52	221	52	0	0	0	0	0	0	0	0	0	1	0	0	2
bst12	39	19	7	7	39	46	5	5	39	53	10	10	39	6	5	5	156	17	6	6	0	0	0	0	0	0	1	1	0	0	1	1	2
bst13	32	32	5	5	57	59	5	5	57	67	6	10	5	5	5	5	5	5	5	5	0	0	0	0	0	0	1	1	0	0	1	1	1
bst14	58	58	4	4	83	86	5	5	83	95	6	10	6	6	4	4	6	6	4	4	0	0	0	0	0	0	1	1	0	0	1	1	1
bst15	32	32	5	5	57	59	5	5	57	67	6	10	5	5	5	5	5	5	5	5	0	0	0	0	0	0	1	1	0	0	1	1	1
bst16	140	92	140	4	106	107	17	16	106	128	26	26	44	26	9	3	50	32	10	4	0	0	0	0	0	0	0	0	0	0	0	0	1
bst17	32	32	5	32	69	56	5	5	69	74	10	10	5	5	5	5	5	5	5	5	0	0	0	0	0	0	1	1	0	0	3	2	3
bst18	32	32	5	5	58	59	5	5	58	68	6	10	5	5	5	5	5	5	5	5	0	0	0	0	0	0	1	1	0	0	1	1	2
bst19	7	19	4	4	28	46	5	29	53	57	10	10	5	6	8	8	13	14	16	16	0	0	0	0	0	0	1	0	0	0	1	1	3
bst20	58	58	4	4	80	81	5	5	80	102	10	10	6	6	4	4	6	6	4	4	0	0	0	0	0	0	2	2	0	0	2	2	2
bst21	71	71	4	4	99	106	5	5	99	114	6	10	12	12	4	4	13	13	4	4	0	0	0	0	0	0	1	1	0	0	1	1	1
bst22	1	40	6	6	5	62	5	5	9	28	10	10	1	32	7	7	5	73	11	11	1	0	0	0	1	0	1	1	1	0	1	1	4
bempl1	19	19	3	3	19	7	10	10	19	17	17	19	19	6	2	2	48	6	2	2	0	0	0	0	0	0	0	0	0	0	0	0	1
cd1	45	37	4	1	45	38	5	5	45	45	10	10	45	11	3	1	58	13	5	1	0	0	0	1	0	0	1	1	0	0	1	1	2
cd2	34	16	1	1	34	25	4	5	34	25	4	6	34	6	1	1	47	9	2	2	0	0	1	1	0	0	1	1	0	0	1	1	1
cd3	22	37	2	1	38	46	5	5	38	46	10	10	11	11	2	1	14	13	2	1	0	0	0	1	0	0	2	2	0	0	2	2	2
ctree1	18	18	39	39	18	5	56	56	18	8	21	21	18	4	61	47	67	4	71	51	0	0	0	0	0	1	0	0	0	1	0	0	1
dll1	4	6	4	4	16	30	19	14	16	33	19	26	3	4	3	3	6	7	6	6	0	0	0	0	0	0	0	0	0	0	0	0	2
dll2	4	6	1	1	16	30	5	5	16	36	10	10	3	4	1	1	7	8	1	1	0	0	1	1	0	0	2	1	0	0	2	2	2
dll3	31	1	1	2	26	5	5	5	26	7	10	10	11	1	1	2	13	1	1	2	0	1	1	0	0	1	2	2	0	1	2	2	3
dll4	4	6	1	1	16	30	4	5	16	33	4	8	3	4	1	1	7	8	1	1	0	0	1	1	0	0	1	1	0	0	1	1	1
dll5	23	39	1	1	53	64	5	5	53	68	6	10	10	11	1	1	20	21	1	1	0	0	1	1	0	0	1	1	0	0	1	1	2
dll6	37	6	11	11	20	30	10	10	20	33	21	28	11	4	5	5	19	12	11	11	0	0	0	0	0	0	0	0	0	0	0	0	2
dll7	4	6	1	1	3	15	5	5	3	21	8	10	3	4	1	1	6	7	2	2	0	0	1	1	1	0	1	1	1	0	1	1	2
dll8	51	6	1	1	24	30	5	5	24	33	7	10	11	4	1	1	20	13	1	1	0	0	1	1	0	0	1	1	0	0	1	1	1
dll9	21	1	2	2	21	5	5	5	21	9	10	10	21	1	2	2	123	4	4	4	0	1	0	0	0	1	1	1	0	1	1	1	1
dll10	1	55	1	1	2	40	5	5	2	40	5	8	1	26	1	1	1	33	2	2	1	0	1	1	1	0	1	1	1	0	1	1	1
dll11	51	6	11	11	20	30	5	5	20	33	6	10	11	4	3	3	20	13	3	3	0	0	0	0	0	0	1	1	0	0	1	1	1
dll12	4	6	4	4	16	30	19	14	16	33	19	26	3	4	3	3	6	7	6	6	0	0	0	0	0	0	0	0	0	0	0	0	1
dll13	30	3	1	1	30	15	5	5	30	27	10	10	30	2	1	1	80	5	2	2	0	0	1	1	0	0	1	1	0	0	1	1	1
dll14	4	6	1	1	3	15	5	5	3	21	8	10	3	4	1	1	6	7	2	2	0	0	1	1	1	0	1	1	1	0	1	1	2
dll15	51	6	11	11	24	30	5	5	24	33	10	10	11	4	3	3	20	13	3	3	0	0	0	0	0	0	1	1	0	0	1	1	2
dll16	4	6	4	4	16	30	19	14	16	33	19	26	3	4	3	3	6	7	6	6	0	0	0	0	0	0	0	0	0	0	0	0	1
dll17	4	6	4	4	3	30	5	5	3	33	8	10	3	4	3	3	6	7	6	6	0	0	0	0	1	0	1	1	1	0	1	1	3
dll18	14	59	14	14	40	43	44	43	40	56	10	10	11	26	11	11	25	33	25	25	0	0	0	0	0	0	0	0	0	0	1	1	4
dll19	11	29	1	1	37	53	5	5	37	60	5	9	8	9	1	1	15	16	1	1	0	0	1	1	0	0	1	1	0	0	1	1	1
dll20	59	18	2	2	50	41	5	5	50	48	10	10	24	6	3	3	25	9	3	3	0	0	0	0	0	0	1	1	0	0	2	1	3
farmer1	17	90	9	9	94	90	26	26	94	79	40	43	13	97	7	7	30	119	12	12	0	0	0	0	0	0	0	0	0	0	0	0	1
fsm1	4	50	1	1	19	49	5	5	19	50	10	10	3	71	1	1	7	91	1	1	0	0	1	1	0	0	1	1	0	0	1	1	2
fsm2	59	59	1	1	59	59	5	5	59	59	10	10	59	69	1	1	87	78	1	1	0	0	1	1	0	0	1	1	0	0	1	1	1
fsm3	1	34	1	1	2	29	4	5	2	30	4	8	1	25	1	1	6	40	6	6	1	0	1	1	1	0	1	1	1	0	1	1	1
fsm4	36	36	36	36	36	31	5	5	36	32	10	10	36	90	3	3	153	144	3	3	0	0	0	0	0	0	1	1	0	0	1	1	2
fsm5	34	34	34	34	34	29	66	43	34	30	10	10	34	35	18	18	107	48	19	19	0	0	0	0	0	0	0	0	0	0	1	1	3
fsm6	50	50	3	3	50	49	11	11	50	50	30	30	50	39	32	26	116	57	35	29	0	0	0	0	0	0	0	0	0	0	0	0	1
fsm7	24	25	25	25	24	15	40	40	24	19	10	10	24	27	27	27	61	33	30	30	0	0	0	0	0	0	0	0	0	0	1	2	2
fsm8	59	59	59	59	59	59	12	12	59	59	10	10	59	69	8	9	88	79	8	9	0	0	0	0	0	0	0	0	0	0	1	1	1
fsm9	8	34	34	34	24	29	34	29	24	30	34	30	5	45	34	45	7	57	86	57	0	0	0	0	0	0	0	0	0	0	0	0	5
grade1	19	1	23	4	19	5	15	5	19	7	25	10	19	1	42	16	56	1	71	16	0	1	0	0	0	1	0	1	0	1	0	1	1
other1	27	27	28	28	27	28	5	5	27	17	7	10	27	14	8	8	60	18	9	9	0	0	0	0	0	0	1	1	0	0	1	1	1
scl1	4	7	11	11	18	21	8	8	18	41	24	17	3	4	5	5	11	12	11	11	0	0	0	0	0	0	0	0	0	0	0	0	1
scl2	4	9	4	4	18	26	5	5	18	47	10	10	3	4	4	4	11	6	5	5	0	0	0	0	0	0	1	1	0	0	1	1	2
scl3	1	3	3	3	4	9	13	12	4	24	10	10	1	2	2	2	4	10	10	10	1	0	0	0	2	0	0	0	2	0	1	1	3
scl4	4	7	11	11	18	21	8	8	18	42	24	17	3	4	5	5	11	12	11	11	0	0	0	0	0	0	0	0	0	0	0	0	1
scl5	4	7	11	11	18	21	8	8	18	42	24	17	3	4	5	5	11	12	11	11	0	0	0	0	0	0	0	0	0	0	0	0	1
scl6	10	32	1	1	45	38	5	5	45	9	7	10	5	35	1	1	7	51	1	1	0	0	1	1	0	0	2	1	0	1	2	2	4
scl7	1	3	1	1	5	9	5	5	5	24	10	10	1	2	1	1	7	10	7	7	1	0	1	1	2	0	1	1	2	0	1	1	3
scl8	4	7	4	4	19	22	23	9	19	47	23	21	3	4	3	3	6	7	6	6	0	0	0	0	0	0	0	0	0	0	0	0	1
scl9	66	9	1	1	66	36	5	5	66	47	10	10	66	4	1	1	166	5	1	1	0	0	1	1	0	0	1	1	0	0	1	1	1
scl10	4	7	11	11	19	22	8	8	19	47	25	17	3	4	5	5	11	12	11	11	0	0	0	0	0	0	0	0	0	0	0	0	1
scl11	8	23	11	11	38	41	10	10	38	70	27	17	6	7	9	9	16	17	15	15	0	0	0	0	0	0	0	0	0	0	0	0	1
scl12	90	7	11	11	29	31	5	5	29	46	10	10	11	4	3	3	20	13	3	3	0	0	0	0	0	0	2	2	0	0	2	2	2
scl13	116	45	1	1	116	63	5	5	116	98	10	10	116	11	1	1	204	18	1	1	0	0	1	1	0	0	1	1	0	0	1	1	1
scl14	9	34	1	1	46	40	5	5	46	10	7	10	5	40	1	1	7	52	1	1	0	0	1	1	0	0	2	1	0	1	2	1	4
scl15	4	7	11	11	22	21	8	8	22	43	24	17	3	4	5	5	11	12	11	11	0	0	0	0	0	0	0	0	0	0	0	0	1
scl16	32	1	60	1	32	5	56	5	32	10	64	10	32	1	63	1	94	1	94	1	0	1	0	1	0	1	0	1	0	3	0	1	4
scl17	4	32	11	11	22	38	11	9	22	9	9	19	3	55	5	5	11	69	11	11	0	0	0	0	0	0	0	0	0	1	1	0	2
scl18	17	74	4	4	4	53	5	5	4	76	10	10	8	12	9	9	10	14	9	9	0	0	0	0	1	0	1	1	1	0	1	1	3
scl19	1	3	1	1	5	12	5	5	9	24	10	10	1	2	1	1	1	4	3	3	1	0	1	1	1	0	1	1	1	0	1	1	3
Avg/Sum	28.6	29.2	11.7	9.2	35.0	39.9	12.1	10.4	36.0	45.6	14.2	14.1	16.0	17.0	8.7	6.0	31.5	24.1	13.1	8.2	12	4	29	32	19	6	68	65	19	12	81	77	162
Win	35	17	62	65	20	10	58	65	22	10	55	36	40	14	62	67	35	13	72	77	12	4	29	32	16	5	58	55	16	7	63	60	

Fig. 9: Effectiveness for Real Faults.

fault for NND and NNDW, respectively. In total, users can find 32, 65 and 77 faults by inspecting the top 1, 5 and 10 nodes, respectively. In terms of **Win**, AlloyFL$_{hy}$ performs the best under NNUD-1, NNUD-5, NND, NNDW and the top-1 metrics. AlloyFL$_{mu}$ performs the best under NNUD-10 and top-5, top-10 metrics. On average, AlloyFL$_{un}$ performs better than AlloyFL$_{co}$ but worse than AlloyFL$_{mu}$, and AlloyFL$_{hy}$ performs the best.

These results show an important tradeoff between the types of faults each technique is best equipped to handle. Notably,

ratio	nnud-1	nnud-5	nnud-10	nnd	nndw	top-1	top-5	top-10
10	40.8	39.0	40.9	37.6	80.1	8.2	23.5	28.0
20	29.0	28.2	31.4	26.3	52.7	15.5	35.4	43.5
30	21.1	21.3	24.9	18.6	35.4	19.2	43.7	52.9
40	16.9	17.1	20.7	13.9	25.9	21.3	50.5	61.2
50	15.1	15.5	19.3	12.0	21.5	23.5	52.9	64.5
60	12.3	13.0	17.2	9.2	15.3	26.3	56.0	67.7
70	11.5	12.5	16.3	8.4	13.1	27.5	57.5	70.4
80	10.7	11.7	15.5	7.5	11.2	28.7	60.8	73.3
90	10.2	11.1	14.9	7.0	10.0	29.4	61.7	75.4
100	**9.2**	**10.4**	**14.1**	**6.0**	**8.2**	**32.0**	**65.0**	**77.0**

Fig. 10: Test Size Impact for AlloyFL$_{hy}$.

AlloyFL$_{co}$ 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$_{mu}$ is more accurate for faults that can be fixed with mutations, e.g. **addr1** and **bst3**. AlloyFL$_{hy}$ performs, on average, better than AlloyFL$_{co}$ and AlloyFL$_{mu}$ because it benefits from the strengths of both AlloyFL$_{co}$ and AlloyFL$_{mu}$. On the other hand, AlloyFL$_{un}$ 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$_{un}$ cannot locate faults accurately. Finally, our results also confirm our hypothesis and show that AlloyFL$_{hy}$ is substantially more effective than AlloyFL$_{un}$ 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$_{hy}$ (with Ochiai formula and AlloyFL$_{mu}$ weight $\lambda = 0.4$). 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$_{hy}$ reaches the best performance under all metrics when using the full test suite. The effectiveness of AlloyFL$_{hy}$ decreases as the test size decreases.

Specifically, AlloyFL$_{hy}$ 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 increases 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$_{hy}$ 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

Model	**Un**	**Co**	**Mu**	**Hy**
addr1	1.4	1.3	8.5	8.4
arr1	0.9	0.9	2.8	2.8
arr2	1.1	1.1	7.9	7.8
arr3	1.1	1.1	7.8	7.7
arr4	1.6	1.6	11.5	10.9
arr5	1.3	1.2	7.7	7.9
arr6	1.1	1.1	6.6	6.5
arr7	1.2	1.1	8.6	8.4
arr8	1.2	1.2	10.7	10.9
arr9	1.3	1.2	10.5	10.1
arr10	0.9	1.0	3.1	3.2
arr11	1.5	1.6	9.5	9.5
bst1	3.1	3.2	41.1	40.5
bst2	3.3	3.2	75.0	75.9
bst3	3.9	3.9	86.4	86.0
bst4	3.4	3.7	85.8	83.2
bst5	3.8	3.9	65.1	64.1
bst6	3.9	4.1	65.0	64.6
bst7	3.5	3.5	47.2	47.9
bst8	3.1	3.1	60.7	60.5
bst9	3.6	3.8	57.7	57.6
bst10	5.1	4.9	114.4	114.8
bst11	3.6	3.5	75.4	75.4
bst12	3.5	3.1	67.2	67.5
bst13	3.5	3.5	63.2	64.5
bst14	3.5	3.4	61.3	62.7
bst15	4.0	3.8	65.7	66.8
bst16	3.7	3.6	72.3	72.1
bst17	3.9	3.9	82.3	83.2
bst18	3.6	3.6	68.1	66.7
bst19	3.5	3.6	57.0	57.4
bst20	4.2	4.0	83.0	80.6
bst21	3.7	3.5	56.3	56.5
bst22	2.9	2.9	52.8	52.6
bempl1	0.9	1.0	3.6	3.5
cd1	0.8	0.8	3.7	3.8
cd2	0.8	0.8	2.0	2.1
cd3	0.8	0.8	3.3	3.3
ctree1	0.9	0.8	6.7	6.7
dll1	1.4	1.4	9.0	9.0
dll2	1.6	1.4	11.6	11.9
dll3	1.5	1.5	10.3	10.0
dll4	1.5	1.6	9.8	9.7
dll5	1.5	1.4	11.6	11.5
dll6	1.7	1.6	11.1	11.3
dll7	1.5	1.5	11.5	11.8
dll8	1.6	1.8	11.3	11.2
dll9	1.1	1.2	13.1	13.6
dll10	1.4	1.4	8.2	8.2
dll11	1.5	1.5	10.2	10.4
dll12	1.4	1.6	8.8	9.0
dll13	1.5	1.4	13.1	13.0
dll14	1.4	1.4	11.4	11.8
dll15	1.7	1.6	15.4	15.5
dll16	1.5	1.5	9.2	9.0
dll17	1.5	1.4	11.4	10.9
dll18	5.0	4.7	134.3	136.4
dll19	1.3	1.4	9.7	9.6
dll20	1.4	1.4	14.0	13.8
farmer1	2.6	2.7	28.2	28.2
fsm1	0.8	0.8	4.7	5.1
fsm2	0.8	0.8	4.9	4.8
fsm3	0.7	0.8	4.8	4.7
fsm4	0.8	0.8	7.2	7.1
fsm5	0.8	0.7	4.7	4.7
fsm6	0.8	0.8	5.1	5.2
fsm7	0.8	0.7	4.3	4.6
fsm8	0.8	0.8	4.9	4.7
fsm9	0.7	0.8	3.1	3.4
grade1	1.3	1.2	6.3	6.0
other1	0.9	0.8	3.1	3.2
scl1	2.4	2.4	24.5	24.1
scl2	2.6	2.5	39.5	39.5
scl3	2.7	2.5	65.5	67.0
scl4	2.3	2.3	24.5	28.2
scl5	2.6	2.6	24.8	27.5
scl6	2.4	2.4	32.2	34.5
scl7	2.6	2.5	47.2	47.3
scl8	2.4	2.5	31.9	31.9
scl9	2.2	2.3	42.3	42.4
scl10	2.7	2.5	29.2	29.3
scl11	2.5	2.7	31.7	31.4
scl12	2.7	2.7	55.8	56.1
scl13	2.3	2.4	57.0	57.2
scl14	2.4	2.5	31.8	33.7
scl15	2.6	2.5	32.0	32.0
scl16	1.4	1.4	11.7	11.4
scl17	2.7	2.5	33.1	34.0
scl18	2.5	2.4	47.7	48.0
scl19	2.4	2.4	41.6	41.6
Avg	**2.1**	**2.1**	30.5	30.7

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$_{mu}$. 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 result 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, 9, 11, 12, 16, 22, 23, 25, 29, 35, 46, 47, 66, 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

[1] Alloy unsat core. http://alloytools.org/quickguide/unsat.html.

[2] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *JSS*, 2009.

[3] R. Abreu, P. Zoeteweij, and A. J. Van Gemund. Spectrum-based multiple fault localization. In *ASE*, 2009.

[4] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *TAICPART-MUTATION*, 2007.

[5] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong. Fault localization using execution slices and dataflow tests. In *ISSRE*, 1995.

[6] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Directed test generation for effective fault localization. In *ISSTA*, 2010.

[7] T.-D. B Le, D. Lo, C. Le Goues, and L. Grunske. A learning-to-rank based fault localization approach using likely invariants. In *ISSTA*, 2016.

[8] B. Baudry, F. Fleurey, and Y. Le Traon. Improving test suites for efficient fault localization. In *ICSE*, 2006.

[9] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for Java. In *ECOOP*, 2005.

[10] N. Eén and N. Sörensson. An extensible sat-solver. In *SAT*, 2003.

[11] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *ICSE*, 1999.

[12] Z. P. Fry and W. Weimer. A human study of fault localization accuracy. In *ICSM*, 2010.

[13] J. P. Galeotti, N. Rosner, C. G. L. Pombo, and M. F. Frias. TACO: Efficient sat-based bounded verification using symmetry breaking and tight bounds. *TSE*, 2013.

[14] D. Gopinath, R. N. Zaeem, and S. Khurshid. Improving the effectiveness of spectra-based fault localization using specifications. In *ASE*, 2012.

[15] S. Hong, B. Lee, T. Kwak, Y. Jeon, B. Ko, Y. Kim, and M. Kim. Mutation-based fault localization for real-world multilingual programs (t). In *ASE*, 2015.

[16] H. N. Hung Nguyen and T. Nguyen. SQL-aware fault localization in PHP-based web applications. *SANER*, 2018.

[17] P. Jaccard. Étude comparative de la distribution florale dans une portion des alpes et des jura. *Bulletin del la Société Vaudoise des Sciences Naturelles*, 1901.

[18] D. Jackson. Alloy: A lightweight object modelling notation. *TOSEM*, 2002.

[19] D. Jackson. The Alloyed joys of software engineering research, 2017.

[20] D. Jeffrey, N. Gupta, and R. Gupta. Fault localization using value replacement. In *ISSTA*, 2008.

[21] Y. Jia and M. Harman. Higher order mutation testing. *IST*, 2009.

[22] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE*, 2005.

[23] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *ICSE*, 2002.

[24] M. Jose and R. Majumdar. Cause clue clauses: Error localization using maximum satisfiability. In *PLDI*, 2011.

[25] F. Keller, L. Grunske, S. Heiden, A. Filieri, A. van Hoorn, and D. Lo. A critical evaluation of spectrum-based fault localization techniques on a large-scale software system. In *QRS*, 2017.

[26] P. S. Kochhar, X. Xia, D. Lo, and S. Li. Practitioners' expectations on automated fault localization. In *ISSTA*, 2016.

[27] T. B. Le, F. Thung, and D. Lo. Theory and practice, do they match? A case with spectrum-based fault localization. In *ICSME*, 2013.

[28] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*. 1999.

[29] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI*, 2005.

[30] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff. Statistical debugging: A hypothesis testing-based approach. *TSE*, 2006.

[31] F. A. Maldonado-Lopez, J. Chavarriaga, and Y. Donoso. Detecting network policy conflicts using Alloy. In *ABZ*, 2014.

[32] S. Maoz, J. O. Ringert, and B. Rumpe. Cd2alloy: Class diagrams analysis using Alloy revisited. In *MODELS*, 2011.

[33] S. Maoz, J. O. Ringert, and B. Rumpe. CDDiff: Semantic differencing for class diagrams. In *ECOOP*, 2011.

[34] D. Marinov and S. Khurshid. TestEra: A novel framework for automated testing of Java programs. In *ASE*, 2001.

[35] W. Masri. Fault localization based on information flow coverage. *STVR*, 2010.

[36] V. Montaghami and D. Rayside. Extending Alloy with partial instances. In *ABZ*, 2012.

[37] V. Montaghami and D. Rayside. Staged evaluation of partial instances in a relational model finder. In *ABZ*, 2014.

[38] S. Moon, Y. Kim, M. Kim, and S. Yoo. Ask the mutants: Mutating faulty programs for fault localization. In *ICST*, 2014.

[39] L. Naish, H. J. Lee, and K. Ramamohanarao. A model for spectra-based software diagnosis. *TSE*, 2011.

[40] T. Nelson, C. Barratt, D. J. Dougherty, K. Fisler, and S. Krishnamurthi. The margrave tool for firewall analysis. In *LISA*, 2010.

[41] T. Nelson, N. Danas, D. J. Dougherty, and S. Krishnamurthi. The power of "why" and "why not": Enriching scenario exploration with provenance. In *FSE*, 2017.

[42] T. Nelson, S. Saghafi, D. J. Dougherty, K. Fisler, and S. Krishnamurthi. Aluminum: principled scenario exploration through minimality. 2013.

[43] M. Papadakis and Y. Le Traon. Metallaxis-fl: Mutation-

based fault localization. *STVR*, 2015.

[44] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *ISSTA*, 2011.

[45] S. Pearson. Evaluation of fault localization techniques. In *FSE*, 2016.

[46] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller. Evaluating and improving fault localization. In *ICSE*, 2017.

[47] M. Renieres and S. P. Reiss. Fault localization with nearest neighbor queries. In *ASE*, 2003.

[48] N. Ruchansky and D. Proserpio. A (not) nice way to verify the openflow switch specification: Formal modelling of the openflow switch using Alloy. *SIGCOMM*, 2013.

[49] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold. Lightweight fault-localization using multiple coverage types. In *ICSE*, 2009.

[50] I. Shlyakhter. Generating effective symmetry-breaking predicates for search problems. *Discrete Appl. Math.*, 2007.

[51] I. Shlyakhter, R. Seater, D. Jackson, M. Sridharan, and M. Taghdiri. Debugging overconstrained declarative models using unsatisfiable cores. In *ASE*, 2003.

[52] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *TON*, 2003.

[53] A. Sullivan, K. Wang, and S. Khurshid. AUnit: A Test Automation Tool for Alloy. In *ICST*, 2018.

[54] A. Sullivan, K. Wang, R. N. Zaeem, and S. Khurshid. Automated test generation and mutation testing for Alloy. In *ICST*, 2017.

[55] A. Sullivan, R. N. Zaeem, S. Khurshid, and D. Marinov. Towards a test automation framework for Alloy. In *SPIN*, 2014.

[56] E. Torlak, F. S.-H. Chang, and D. Jackson. Finding minimal unsatisfiable cores of declarative specifications. In *FM*, 2008.

[57] E. Torlak and D. Jackson. Kodkod: A relational model finder. In *TACAS*, 2007.

[58] K. Wang, A. Sullivan, and S. Khurshid. Automated model repair for Alloy. In *ASE*, 2018.

[59] K. Wang, A. Sullivan, and S. Khurshid. MuAlloy: A mutation testing framework for Alloy. In *ICSE*, 2018.

[60] K. Wang, A. Sullivan, M. Koukoutos, D. Marinov, and S. Khurshid. Systematic generation of non-equivalent expressions for relational algebra. In *ABZ*, 2018.

[61] K. Wang, A. Sullivan, D. Marinov, and S. Khurshid. Solver-based sketching Alloy models using test valuations. In *ABZ*, 2018.

[62] W. Wang, K. Wang, M. Gligoric, and S. Khurshid. Incremental analysis of evolving Alloy models. In *TACAS*, 2019.

[63] W. Wang, K. Wang, M. Zhang, and S. Khurshid. Learning to optimize the Alloy analyzer. In *ICST*, 2019.

[64] W. E. Wong, V. Debroy, R. Gao, and Y. Li. The DStar method for effective software fault localization. *IEEE Transactions on Reliability*, 2014.

[65] W. E. Wong, V. Debroy, and D. Xu. Towards better fault localization: A crosstab-based statistical approach. In *IEEE Transactions on SMC*, 2012.

[66] X. Xia, L. Gong, T.-D. B. Le, D. Lo, L. Jiang, and H. Zhang. Diversity maximization speedup for localizing faults in single-fault and multi-fault programs. *ASE*, 2016.

[67] X. Xie, F.-C. Kuo, T. Y. Chen, S. Yoo, and M. Harman. Provably optimal and human-competitive results in sbse for spectrum based fault localisation. In *SSBSE*, 2013.

[68] J. Xuan and M. Monperrus. Learning to combine multiple ranking metrics for fault localization. In *ICSME*, 2014.

[69] J. Xuan and M. Monperrus. Test case purification for improving fault localization. In *FSE*, 2014.

[70] P. Zave. Chord alloy model. http://www.pamelazave.com/correctChord.als, 2017.

[71] A. Zeller. Yesterday, my program worked. today, it does not. why? In *FSE*, 1999.

[72] M. Zhang, X. Li, L. Zhang, and S. Khurshid. Boosting spectrum-based fault localization using pagerank. In *ISSTA*, 2017.

[73] D. Zou, J. Liang, Y. Xiong, M. Ernst, and L. Zhang. An empirical study of fault localization families and their combinations. *TSE*, 2019.