Automated Model Repair for Alloy

Kaiyuan Wang
University of Texas at Austin, USA
kaiyuanw@utexas.edu

Allison Sullivan*
University of Texas at Austin, USA
allisonksullivan@utexas.edu

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

ABSTRACT

Automated program repair is an active research area. However, existing research focuses mostly on imperative code, e.g. in Java. In this paper, we study the problem of repairing declarative models in Alloy – a first order relational logic with transitive closure. We introduce ARepair, the first technique for repairing Alloy models. ARepair follows the spirit of traditional automated program repair techniques. Specifically, ARepair takes as input a faulty Alloy model and a test suite that contains some failing test, and outputs a repaired model that is correct with respect to the given tests. ARepair integrates ideas from mutation testing and program synthesis to provide an effective solution for repairing Alloy models. The experimental results show that ARepair can fix 28 out of 38 real-world faulty models we collected.

CCS CONCEPTS

• **Software and its engineering** → *Error handling and recovery*;

KEYWORDS

Model repair, first-order logic, ARepair

ACM Reference Format:
Kaiyuan Wang, Allison Sullivan, and Sarfraz Khurshid. 2018. Automated Model Repair for Alloy. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3238147.3238162

1 INTRODUCTION

Automatic program repair techniques significantly reduce the human effort required to diagnose, debug, and repair faulty programs [2, 7, 17, 21, 24, 30, 35, 38, 39, 43, 59, 63, 66, 79]. The standard *generate-and-validate* approach [14, 19, 29, 34, 62, 80, 81, 83, 84] starts with a faulty program and a test suite that reveals the defect. It explores candidate programs in the search space, and validates each candidate program against the given test suite until a program that passes all tests is found. Some repair techniques infer specs of the program and translate the repair problem into constraints, and then use SAT/SMT solvers to synthesize patches that conform to the specs [33, 44, 55, 64].

[*]Now at North Carolina A&T State University.

Existing program repair techniques mainly focus on imperative languages like Java. Our focus in this paper is declarative models written in Alloy [22] – a first-order relational logic with transitive closure. The Alloy language and its back-end analyzer have been explored within the software engineering community. Alloy is used in various domains, including UML analysis [31, 41, 42], security [40, 52], networking [61], and feature modeling [20]. Additionally, the Alloy analyzer has been extended to provide better scenario finding experiences [6, 28, 47, 49, 54]. Alloy users write models that describe the properties of the system of interest. The Alloy analyzer translates Alloy models into Kodkod [73] formulas and invokes off-the-shelf SAT solvers to search for solutions. The analyzer performs *scope-bounded* analysis, which checks the properties within a given *scope*, i.e. bound on the universe of discourse. AUnit [69] defines the notion of testing in Alloy following the spirit of traditional testing frameworks, e.g. JUnit. Developers write test predicates and invoke commands to assert the existence or non-existence of solutions.

In this paper, we present ARepair, a novel generate-and-validate program repair technique for Alloy, which is able to handle Alloy models with multiple faults. ARepair has three main components: (1) a mutation-based fault localization technique [48, 56], AlloyFL [77] that locates faults at the AST node granularity; (2) a generator that systematically generates Alloy expressions (with equivalence pruning rules for relational algebra [75]); and (3) a synthesizer that explores the search space until a model with all passing tests is found. ARepair starts by invoking AlloyFL to locate faults. Each time AlloyFL is invoked, ARepair checks if the most suspicious node can be fixed by mutation and applies the change if that is the case. Otherwise, for each suspicious AST node returned, ARepair creates holes for descendant nodes in the suspicious AST and enumerates candidate fragments (generated by the expression generator) of corresponding holes until some failing test passes and the results of passing tests are preserved. ARepair implements two strategies to explore the search space: *all-combinations* and *base-choice*. The all-combinations strategy explores all combinations of candidate fragments for all holes until some failing test passes and no passing test fails. The base-choice strategy enumerates candidate fragments for one hole at a time, while keeping the candidate fragments of the other holes constant. After enumerating all the fragments for one hole, the base-choice strategy fills the hole with the fragment that makes the most failing tests pass and no passing test fails. Both strategies are inspired by textbook input space criteria for test coverage [4]. ARepair avoids running AUnit test predicates with expensive SAT solver calls by building formula dependency graphs and leveraging Alloy's built-in evaluator to evaluate a minimal number of affected formulas that determine a test's satisfiability. A hierarchical cache further reduces the sizes of inputs to the evaluator. ARepair repeatedly fixes faults until all tests pass or it exhausts the bounded search space.

We evaluate ARepair using models collected from the standard Alloy 4.1 distribution and Amalgam [53]. We also collect Alloy assignment solutions from graduate students. With the default setting, ARepair is able to repair 28 out of 38 *real faulty models*. We make the following contributions:

- **Alloy Model Repair.** ARepair is the first repair technique for Alloy, which uses both mutations and synthesis to repair faulty models. The experimental results show that the combined approach works well and many faulty models require both mutations and synthesis for a complete fix. ARepair does not require isolated faults. It can fix models with multiple faults or faults involving multiple locations.
- **Optimizations for Practical Model Repair.** ARepair does not search for fault patterns and apply repair templates to fix faults. Instead, it tries to repair a faulty AST in a bottom-up fashion, so it is more likely to repair faults with unseen patterns. The absence of repair templates results in an immense search space and we implement the following optimizations to make the technique tractable and reduce end-to-end time. (1) The expression generator prunes equivalent expressions based on equivalence pruning rules [75] and modulo test inputs [32]. (2) The enumeration based approach explores the search space without expensive constraint solving [3]. (3) The construction of dependency graphs for constraint formulas enables a small number of evaluator calls during the enumeration based approach. (4) The base-choice search strategy reduces the exploration space. (5) The hierarchical caching reduces sizes of the inputs to evaluator calls.
- **Evaluation.** We evaluate ARepair on real faults and show that it is able to fix 28 out of 38 faulty models. We qualitatively compare patches generated by ARepair and human written patches, and show that the quality of the generated patches is good.
- **Open Source.** We release 38 real-world Alloy models with annotated fault locations and human-written patches, and open source ARepair so researchers can use them in the future. The repo is available at https://github.com/kaiyuanw/ARepair.

2 EXAMPLE

This section presents a real-world faulty Alloy model to introduce the basics of Alloy and AUnit. Then, we describe how AlloyFL [77] and ARepair fix the model.

Figure 1a shows the "Farmer River Crossing" puzzle where the goal is to allow a farmer to transport a fox, chicken and grain from one river bank to the other. The farmer uses a boat and can only carry one item at a time. If left unattended, the fox eats the chicken and the chicken eats the grain. The model contains a fault which prevents the "eating" from happening while the farmer is away. Instead, the faulty model enforces the "eating" to happen when the farmer comes back.

Lines 2-4 declare the basic types in the problem: a notion of object (line 2; `sig` denotes a set and introduces a type), four concrete objects (line 3), and a set of states (line 4) that model the objects in both the `near` and `far` banks after every farmer's river crossing action (line 4). The `abstract` keyword enforces that an object is one of its concrete subtypes: `Farmer`, `Fox`, `Chicken` and `Grain`. The `one` keyword constrains each concrete object type to contain a single, distinct object atom. The `eats` field declares that each object can eat a set of objects and the fact on line 5 restricts that the fox can eat the chicken and the chicken can eat the grain. The `initialState` fact on lines 6-7 constrains that initially everything is on the near bank and nothing is on the far bank. The `crossRiver` predicate on lines 8-13 defines the river crossing action. It takes four parameters: the set of objects on the bank where the farmer starts at (pre-state: `from` and post-state: `from'`) and the set of objects on the bank the farmer will cross to (pre-state: `to` and post-state: `to'`). The predicate states that either the farmer takes nothing or the farmer takes one item to the other side of the river. The `stateTransition` fact on lines 14-19 states that for every two consecutive states, if the farmer is on the near bank in the pre-state, then he would cross the river to the far bank in the post-state, and vice versa. The `solvePuzzle` on line 20 restricts that everything should be on the far bank of the river in the last state.

The fault is in the `crossRiver` predicate (highlighted in orange). The predicate enforces eating to happen only after the farmer comes back and not immediately after the farmer leaves the bank. This means if the farmer takes the grain from the near bank to the far bank, the fox will not eat the chicken. But when the farmer comes back to the near bank, the fox eats the chicken. This modeling error was in Alloy release 4.1 and was fixed in release 4.2.

An AUnit test [69] that reveals the fault is shown in Figure 1b. Predicate `test1` encodes the valuations of each signature type and field relation in the faulty farmer model. The invocation of `crossRiver` predicate on line 18 states that everything is on one bank and nothing is on the other bank in the pre-state. In the post-state (after the farmer crosses the river with the fox), only the chicken is left on the one bank (because the chicken eats the grain) and both the farmer and the fox are on the other bank. The command in line 19 runs the test with at most 4 atoms for each sig type and expects the existence of a solution. However, the test predicate is unsatisfiable because of the modeling error, resulting in a test failure.

ARepair invokes AlloyFL to locate faults at the AST node granularity. The most suspicious node AlloyFL returns is shown in Figure 2. ARepair creates holes to replace each level of AST nodes in a bottom-up fashion. For example, it first creates holes for `to` and `eats` (highlighted in red). Then, ARepair generates a set of candidate expressions for each hole using all signatures/fields/variables in scope, e.g. `Farmer`, `from` and `item`, etc. Next, ARepair enumerates the candidate expressions for each hole and runs all affected tests to see if any test result changes from failing to passing. ARepair keeps the candidate values that make some failing tests pass and preserves the results of passing tests. In this case, ARepair replaces `to` with `none` and now one failing test passes (and no passing test fails). Next, ARepair reruns AlloyFL and finds that the most suspicious node is still the same. In this iteration, ARepair creates holes for `to` and the relational join operator "." (highlighted in yellow). ARepair keeps synthesizing expressions/formulas under each suspicious node to make failing tests pass. If ARepair cannot make any failing test pass for the suspicious node, then it repeats the same process for the next suspicious node. Note that AlloyFL is a mutation-based technique and it can also repair the model with mutations. Each time AlloyFL is invoked, we check if there is a mutation over the most suspicious node AlloyFL reports that makes some failing tests pass and no passing test fails. If such mutation

```
 1. open util/ordering[State] as ord
 2. abstract sig Object { eats: set Object }
 3. one sig Farmer, Fox, Chicken, Grain extends Object {}
 4. sig State { near,far: set Object }
 5. fact eating { eats = Fox->Chicken + Chicken->Grain }
 6. fact initialState {
 7.   let s0 = ord/first | s0.near = Object && no s0.far }
 8. pred crossRiver[from,from',to,to': set Object] {
 9.   (from' = from - Farmer
10.     && to' = to - to.eats + Farmer ) ||
11.   (some item: from - Farmer {
12.     from' = from - Farmer - item
13.       && to' = to - to.eats + Farmer + item })}
14. fact stateTransition {
15.   all s: State, s': ord/next[s] {
16.     Farmer in s.near =>
17.       crossRiver[s.near, s'.near, s.far, s'.far]
18.       else crossRiver[s.far, s'.far, s.near, s'.near]
19.   }}
20. pred solvePuzzle { ord/last.far = Object }
```

(a) Faulty farmer river crossing model.

```
 1. pred test1 {
 2.   some disj F0: Farmer | some disj X0: Fox |
 3.   some disj C0: Chicken | some disj G0: Grain |
 4.   some disj F0, X0, C0, G0: Object |
 5.   some disj S0, S1, S2, S3: State {
 6.     Farmer = F0
 7.     Fox = X0
 8.     Chicken = C0
 9.     Grain = G0
10.     Object = F0 + X0 + C0 + G0
11.     eats = X0->C0 + C0->G0
12.     State = S0 + S1 + S2 + S3
13.     near = S0->F0 + S0->X0 + S0->C0 + S0->G0
14.            + S1->X0 + S2->F0 + S2->X0 + S3->X0
15.     far = S1->F0 + S1->G0 + S2->G0 + S3->F0 + S3->G0
16.     ord/first = S0
17.     ord/next = S0->S1 + S1->S2 + S2->S3
18.     crossRiver[F0+X0+C0+G0, C0, none, F0+X0] } }
19. run test1 for 4 expect 1
20. // More tests ...
```

(b) A failing test of the farmer river crossing model [77].

Figure 1: Faulty Farmer Example and Tests.

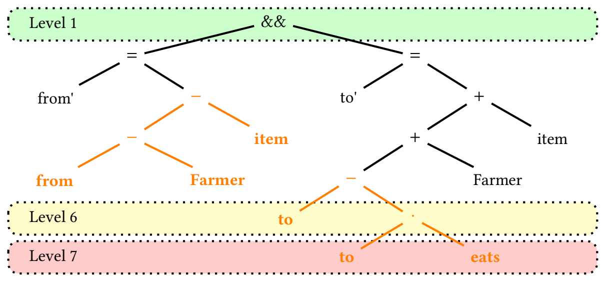

Figure 2: First Suspicious Node for Faulty Farmer Example.

(A) A human-written patch.

```
 1.   pred crossRiver[from,from',to,to': set Object] {
 2.-    (from' = from - Farmer
 3.+    (from' = from - Farmer - from'.eats
 4.-      && to' = to - to.eats + Farmer ) ||
 5.+      && to' = to + Farmer ) ||
 6.     (some item: from - Farmer {
 7.-      from' = from - Farmer - item
 8.+      from' = from - Farmer - item - from'.eats
 9.-        && to' = to - to.eats + Farmer + item })}
10.+        && to' = to + Farmer + item })}
```

(B) A patch generated by ARepair.

```
 1.   pred crossRiver[from,from',to,to': set Object] {
 2.-    (from' = from - Farmer
 3.-      && to' = to - to.eats + Farmer ) ||
 4.-    (some item: from - Farmer {
 5.+    (some item: from + Farmer {
 6.-      from' = from - Farmer - item
 7.+      from' = from - (Farmer + from'.eats) - item
 8.-        && to' = to - to.eats + Farmer + item })}
 9.+        && to' = to + Farmer + item })}
```

Figure 3: Patches for the faulty farmer model.

exists, then we mutate the model and start the next iteration. Finally, if ARepair can fix the faulty model, i.e. all tests pass, then it post-processes the fixed model to remove redundant code, e.g. replace "`to-none.eats`" with "`to`", and returns the model to the user.

Figure 3 shows the human written patch (A) and the first patch generated by ARepair (B). We can see that the human written patch fixes the "eating" action both when the farmer crosses the river with (lines 7-10) or without (lines 2-5) an item. The patch ARepair generates deletes the formula that models the farmer's crossing-river without an item (lines 2-3), and fixes the "eating" action when the farmer crosses the river with an item (lines 6-9). The interesting part is that the patch also changes the domain of a variable declaration (lines 4-5), which actually merges both cases when the farmer crosses the river with/without an item. The new domain (line 5) allows the `item` to be the farmer himself and it models the correct semantics corresponding to the deleted formula on lines 2-3. In this case, we validate the equivalence of generated patch and the human-written patch with a *scope-bounded* analysis using the Alloy analyzer and find that the generated patch is semantically equivalent to the human written patch.

3 BACKGROUND: FAULT LOCALIZATION

In this section, we describe AlloyFL [77], which is the fault localization technique used by ARepair.

AlloyFL follows the traditional mutation-based fault localization techniques [48, 56] and implements a variety of mutation operators as shown in Figure 4. *MOR* mutates signature multiplicity, e.g. "`lone sig`" to "`one sig`". *QOR* mutates quantifiers, e.g. "`all`" to "`some`". *UOR*, *BOR* and *LOR* define operator replacement for unary, binary and formula 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 an Alloy 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`". These operators are well defined such that AlloyFL is able to accurately locate faults and even fix the faults in some cases. For example, to fix the faulty farmer example in Figure 1a, AlloyFL fixes parts of the faults by applying *BOR* (Figure 3 (B) lines 4-5) and *LOD* (Figure 3 (B) lines 2-3).

The input of AlloyFL is a faulty Alloy model and a set of Alloy commands with the `expect` keyword. These commands can invoke Alloy predicates, function or assertions. "`expect 1`" means that the corresponding command is expected to be satisfiable while "`expect`

Mutation Operator	**Description**
MOR	Multiplicity Operator Replacement
QOR	Quantifier Operator Replacement
UOR	Unary Operator Replacement
BOR	Binary Operator Replacement
LOR	Formula 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

Figure 4: Mutation Operators

`0`" means that the command is expected to be unsatisfiable. In this paper, each command invokes an AUnit [69] test predicate and we say an AUnit test fails if the corresponding command is satisfiable but is expected to be unsatisfiable, or vice versa. The output of AlloyFL is a list of AST nodes in descending order of their suspiciousness given a formula. In this paper, we use the Ochiai [1] formula $\frac{failed(e)}{\sqrt{totalfailed \times (failed(e)+passed(e))}}$, where *failed(e)* and *passed(e)* are the number of tests that failed and passed (with respect to the original faulty model) that kill the mutant *e*, and *totalfailed* is the total number of failed tests for the faulty model. AlloyFL systematically mutates the faulty Alloy model using mutation operators in Figure 4 and runs the test suite against each mutant. A suspiciousness score computed from the Ochiai formula is assigned to each mutated AST node. In case more than one mutation operator is applicable to an AST node, the maximum suspiciousness score computed for the node is used. Finally, AlloyFL ranks all nodes in the descending order of suspiciousness and returns the ranked list. We modify AlloyFL to also return the mutation operator corresponding to the most suspicious AST node so later ARepair can determine if that mutation should be applied as a potential fix.

4 TECHNIQUE

In this section, we first describe how we create holes (Section 4.1) and how we generate expressions to fill in holes (Section 4.2). Next, we describe the search strategies (Section 4.3). Then, we describe how we run tests without invoking a SAT solver (Section 4.4) and the hierarchical caching we use to improve performance (Section 4.5). Finally, we describe the enumeration-based repair approach as a whole (Section 4.6).

4.1 Create Holes

For each suspicious AST node returned by AlloyFL, we create holes at each level of the corresponding AST in a bottom-up fashion. For example, the most suspicious node in the faulty farmer model (Figure 2) has 7 levels. We first create holes at level 7 (shown in red) and synthesize new expressions at that level without modifying nodes of other levels. We repeat this process from level 7 to level 1 (root level) until some failing test passes and no passing test fails. The intuition is that AlloyFL is designed to mutate upper-level operator nodes and if the fault cannot be fixed by AlloyFL, then the issue is likely at the lower levels of the AST. This approach also prioritizes patches with smaller perturbations to the original model, which is consistent with the insight: patches that introduce smaller perturbations to the original program are more likely to be correct [10, 33].

Creating a single hole for each node in a given level may not result in valid models. For example, replacing the `&&` node with a hole at level 1 in Figure 2 does not make the new program compile. Consequently, the schema to create holes for different AST nodes may vary. ARepair introduces different types of holes, i.e. quantifier holes (denoted by *qh*), logical operator holes (denoted by *loh*), comparison operator holes (denoted by *coh*), implication holes (denoted by *ih*), cardinality holes (denoted by *ch*), boolean holes (denoted by *bh*) and expression holes (denoted by *eh*). The value of *qh* can be one of "`all`", "`no`", "`some`", "`lone`" or "`one`". The value of *loh* can be either "`&&`" or "`||`". The value of *coh* can be one of "`=`", "`in`", "`!=`" or "`!in`". The value of *ih* can be either "`=>`" or "`<=>`". The value of *ch* can be one of "`no`", "`lone`", "`one`" or "`some`". The value of *bh* can be either empty ϵ or "`!`". The value of *eh* can be any expression.

Figure 5 shows the meaning of different types of AST nodes and the corresponding schemas to create holes. Each schema is denoted by "$\bar{h}(x) := H$", where $\bar{h}(x)$ means the holes created from AST node type x and H shows the way to compute holes. For example, the most suspicious node returned by AlloyFL is a conjunction node ("`&&`" as shown in Figure 2). To create holes for the root node, we can apply the schema for the conjunction node, which states that the holes we should create include a logical operator hole *loh*, holes created from the left child and holes created from the right child. In this case, both the left and right children of the conjunction node are set equality nodes ("`=`"), so we recursively apply schemas in Figure 5 until no more holes are created. In the end, we would create 4 expression holes, 2 comparison operator holes and a logical operator hole. This step guarantees that if we fill holes with candidate operators/expressions, then the resulting expression/formula will always compile.

4.2 Generating Expressions

The space of candidate fragments for operator holes, e.g. quantifier holes and cardinality holes, are fixed, but the space of candidate fragments for expression holes depends on the expression generator. To generate valid candidate fragments for expression holes, we need to find all atomic expressions in the model that can be used. ARepair has a static analyzer which finds all atomic expressions, i.e. sigs, fields, predicate/function parameters, quantifier variables and let variables, in scope of each expression hole. The holes that share the same set of atomic expressions in scope have the same set of generated candidate fragments.

ARepair implements an expression generator that generates expressions following the grammars in Figure 6. The generator implements two pruning strategies. First, the generator prunes semantically equivalent expressions using equivalence pruning rules for relational algebra described in RexGen [75]. Second, the generator implements a modulo test checker that prunes equivalent expressions with respect to the given tests. The equivalence pruning rules and the modulo test checker significantly reduce the number of expressions to consider and make the repair problem tractable.

Meaning	Schema	Meaning	Schema
Cartesian product	$\bar{h}(\phi \times \psi) := eh$	Relational join	$\bar{h}(\phi \bowtie \psi) := eh$
Union	$\bar{h}(\phi \cup \psi) := eh$	Intersection	$\bar{h}(\phi \cap \psi) := eh$
Set difference	$\bar{h}(\phi \setminus \psi) := eh$	Overriding union	$\bar{h}(\phi{+}{+}\psi) := eh$
Domain restriction	$\bar{h}(\phi <: \psi) := eh$	Range restriction	$\bar{h}(\phi :> \psi) := eh$
Transitive closure	$\bar{h}(\hat{}\phi) := eh$	Reflexive transitive closure	$\bar{h}(*\phi) := eh$
Inverse relational join	$\bar{h}(\phi[\psi]) := eh$	Relational transpose	$\bar{h}(\sim\phi) := eh$
Cardinality	$\bar{h}(\#\phi) := eh$	Set comprehension	$\bar{h}(\{\bar{t} : \phi \vert \alpha(\bar{t})\}) := eh$
Identity relation (binary)	$\bar{h}(iden) := eh$	Universe (unary)	$\bar{h}(univ) := eh$
Conjunction	$\bar{h}(\alpha \wedge \beta) := \bar{h}(\alpha)\ loh\ \bar{h}(\beta)$	Disjunction	$\bar{h}(\alpha \vee \beta) := \bar{h}(\alpha)\ loh\ \bar{h}(\beta)$
Implication	$\bar{h}(\alpha \Rightarrow \beta) := \bar{h}(\alpha)\ ih\ \bar{h}(\beta)$	Bi-implication	$\bar{h}(\alpha \Leftrightarrow \beta) := \bar{h}(\alpha)\ ih\ \bar{h}(\beta)$
If-then-else	$\bar{h}(\alpha?\beta : \gamma) := \bar{h}(\alpha)\ \bar{h}(\beta)\ \bar{h}(\gamma)$	Negation	$\bar{h}(\neg\alpha) := bh\ \bar{h}(\alpha)$
Relational equality	$\bar{h}(\phi = \psi) := \bar{h}(\phi)\ coh\ \bar{h}(\psi)$	Relational containment	$\bar{h}(\phi\ in\ \psi) := \bar{h}(\phi)\ coh\ \bar{h}(\psi)$
Universal quantification	$\bar{h}(\forall\bar{t} : \phi \vert \alpha(\bar{t})) := qh\ \bar{h}(\phi)\ \bar{h}(\alpha(\bar{t}))$	Existential quantification	$\bar{h}(\exists\bar{t} : \phi \vert \alpha(\bar{t})) := qh\ \bar{h}(\phi)\ \bar{h}(\alpha(\bar{t}))$
$\vert\phi\vert = 1$	$\bar{h}(one\ \phi) := ch\ \bar{h}(\phi)$	$\vert\phi\vert \leq 1$	$\bar{h}(lone\ \phi) := ch\ \bar{h}(\phi)$
$\vert\phi\vert \geq 1$	$\bar{h}(some\ \phi) := ch\ \bar{h}(\phi)$	$\vert\phi\vert = 0$	$\bar{h}(no\ \phi) := ch\ \bar{h}(\phi)$

Figure 5: Hole creation schemas for Alloy Surface Syntax. $\bar{h}(x)$ computes the holes for syntax x. α, β and γ denote formulas which evaluate to true or false. ϕ and ψ denote expressions which evaluate to relations. $\bar{t} : \phi$ denote tuple membership $\bar{t} \in \phi$.

expression	$e :=$	$uop\ e \mid e\ bop\ e \mid atom_e \mid const$
unary op	$uop :=$	$* \mid \hat{} \mid \sim$
binary op	$bop :=$	$+ \mid \& \mid - \mid \cdot$
atomic expr	$atom_e :=$	$sig_e \mid field_e \mid param_e \mid var_e$
constant	$const :=$	$none \mid iden \mid univ \mid Int \mid 0 \mid 1$

Figure 6: Expression generation syntax.

Next, we describe the modulo test pruning technique with an example. Consider the model shown below:

```
sig Node { link: lone Node } pred p { some n: Node | no ?? }
pred t1 { some disj N0: Node | Node=N0 && link=N0->N0 }
pred t2 { some disj N0,N1: Node {
  Node=N0+N1 && link=N0->N1+N1->N0 && p[] } }
```

The model has a signature `Node`, a binary field `link` and a quantifier variable `n`. Implicitly, "n" is of type `Node` and has a cardinality of 1. The model contains two AUnit tests `t1` and `t2`. Suppose we want to generate expressions of type `Node` in the body (denoted by ??) of the existential quantification, and we can use "n", "link" and "Node" as the atomic expressions. The following table shows the valuations of four syntactically different expressions, i.e. "n", "`n.link`", "`link.(Node-n)`" and "`(link.Node)&n`", with respect to `t1` and `t2`.

test	n	n	n.link	link.(Node-n)	(link.Node) & n
t1	N0	{N0}	{N0}	∅	{N0}
t2	N0	{N0}	{N1}	{N0}	{N0}
	N1	{N1}	{N0}	{N1}	{N1}

For test `t1`, `n` can only be `N0` and `link` is `N0->N0`. It is easy to see that "n", "`n.link`" and "`(link.Node)&n`" evaluate to the same set {`N0`} and thus are equivalent with respect to `t1`. For test `t2`, `n` can be `N0` or `N1`, and `link` is {`N0->N1`, `N1->N0`}. "n", "`link.(Node-n)`" and "`(link.Node)&n`" are equivalent with respect to `t2` both when `n=N0` and `n=N1`. So "n" and "`(link.Node)&n`" are equivalent with respect to the test suite and the modulo test checker can prune either "n" or "`(link.Node)&n`". In practice, we keep expressions with smaller sizes, so "`(link.Node)&n`" will be pruned. If the expression does not contain any free variable, its valuation does not change based on the valuations of free variables. If the expression contains more than one free variable, then we need to enumerate all combinations of possible valuations of the free variables to get the valuations of the expression. If the free variable's cardinality is greater than 1, then its valuation can be any subset of its declaring type. Two expressions are equivalent with respect to a test if their valuations are the same across all combinations of possible valuations of free variables in the scope. The expression generator prunes expressions that are equivalent to any existing expression with respect to the entire test suite.

In this paper, the size of an expression is defined as the number of descendant nodes in the AST representation of the expression. The expression generator is able to generate expressions of a given type and size.

4.3 Search Strategies

Given a level of nodes in a suspicious node and the corresponding holes created, ARepair implements two search strategies: *all combinations* and *base choice* [4].

All combinations. Under this search strategy, ARepair tries all combinations of candidate fragments for all holes until it finds some failing test passed and no passing test failed. This strategy is typically impractical as the number of holes and the number of candidate fragments for each hole grow. For example, with 4 holes and 100 candidate fragments for each hole, the search space is 10^8. In our implementation, we limit the maximum number of combinations to explore (per level of holes) for this search strategy. Typically, the limit we set is still large, so we stop exploring more combinations *the first time* a combination of candidate fragments makes some failing test pass and no passing test fails. If such a combination is found, we fill the holes with the corresponding fragments and save the changes before starting the next iteration.

Intuitively, we want to first explore combinations of candidate fragments of expression holes with smaller expression sizes, because we assume small-sized expressions are more natural to developers, e.g. "n" vs "`(Node-*link.n)`". Figure 7 shows how we prioritize exploring combinations of candidate expressions with smaller sizes. Suppose we have n holes ($hole_1$ to $hole_n$) and $hole_i$ has S_i number of candidate fragments. Then we can partition the candidate fragments of $hole_i$ into k_i parts (P^i_1 to $P^i_{k_i}$) of equal size. Next, we create size-n tuples $U = \{(x_1, x_2, \ldots, x_n) \mid \bigwedge_{i=1}^{n} x_i \in [1, k_i]\}$ and

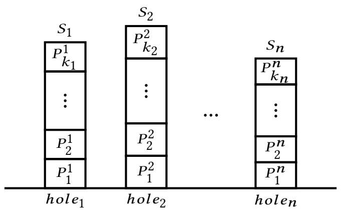

Figure 7: All combinations partitions.

sort the tuples first by $\sum_{i=1}^{n} x_i$ and then by $|\max_{\forall i \in [1,n]} x_i - \min_{\forall i \in [1,n]} x_i|$ to gets a ranked list of tuples L. For example, if $n = 2, k_1 = 2, k_2 = 3$, then the ranked tuple list $L = [(1, 1), (1, 2), (2, 1), (2, 2), (1, 3), (2, 3)]$. Finally, we iterate each tuple $(x_1, \ldots, x_n)$ in L and explore all combinations of candidate fragments $C = \{(f_1, \ldots, f_n) \mid \bigwedge_{i=1}^{n} f_i \in P^i_{x_i}\}$. Since expressions are generated in a bottom-up fashion, expressions with smaller sizes are generated first, which means expression sizes in P^x_i are smaller than expression sizes in P^x_j if $i < j$. Therefore, the exploration strategy guarantees that combinations of smaller expressions are explored first.

Base Choice. Under this search strategy, ARepair holds candidate fragments of all holes constant except one hole (base choice). It enumerates candidate fragments of $hole_i$ with the candidate fragments of the rest holes unchanged. For each $hole_i$, ARepair explores all candidate fragments and picks the one (f_i) that makes the maximum number of failing tests pass and no passing test fails. Then, ARepair enumerates candidate fragments of $hole_{i+1}$ with the fragment of $hole_i$ set to f_i. ARepair uses this exploration strategy from $hole_1$ to $hole_n$ and saves the final changes as the potential fix. For example, with 4 holes and 100 candidate fragments for each hole, the search space is 400. In practice, the number of generated candidate fragments for an expression hole can be large, so we set a limit on the number of candidate fragments to explore per hole.

4.4 Running Tests

ARepair invokes tests in the expression generation phase (to prune expressions), the fault localization phase (to locate faults) and the repair phase (to validate candidate patches). Since the search space is large and each repair problem contains many tests, invoking all tests at the repair phase takes a majority of the time. Moreover, invoking each test predicate with a SAT solver is expensive. We introduce a technique that determines test satisfiability using Alloy's built-in evaluator (without sat solving) and builds a dependency graph for each test to reduce the number of evaluator calls.

For a given faulty Alloy model, ARepair normalizes the signature multiplicity constraints, the field multiplicity constraints and the signature facts, and creates a formula for each constraint. In the faulty farmer example (Figure 1a), the `Object` signature is declared to be an arbitrary set of atoms, so it does not need to be normalized and we create an empty formula (denoted by $\texttt{Object}_{mult}$) which evaluates to true by default. Similarly, the field `eats` is declared to relate an object to a set of objects, so we simply create an empty formula (denoted by $\texttt{eats}_{mult}$). "`one sig Farmer`" is declared to be a singleton set so we normalize it as "`sig Farmer`" (remove signature multiplicity constraint) and create a formula "`one Farmer`" (denoted by $\texttt{Farmer}_{mult}$). Thus, ARepair creates a formula for each signature and field. Since the farmer model does not have any signature facts, we do not need to create any formula for signature facts. For each fact paragraph, ARepair creates a formula (denoted by the fact name) that is identical to the fact body.

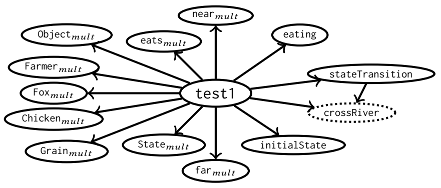

Figure 8: Dependency graph for `test1` in Figure 1b.

For each AUnit test, we create a dependency graph that encodes the formulas the test depends on. For example, Figure 8 shows the dependency graph for `test1` in Figure 1b. `test1` depends on all signature/field multiplicity constraints and all fact constraints, because those constraints are enforced by the Alloy analyzer when we invoke the test. Since both `test1` and `stateTransition` directly invoke the `crossRiver` predicate, they both depend on `crossRiver`.

Once we build the dependency graph for each AUnit test, it is easy to compute a test's satisfiability from the formulas the test depends on. Initially, ARepair evaluates each formula the test depends on and stores the satisfiability of each formula. When ARepair enumerates candidate fragments for holes, it only evaluates the affected formulas to determine the satisfiability of the test. In the faulty farmer example, when ARepair enumerates candidate fragments for holes under the most suspicious AST node (Figure 2), the only affected predicate is `crossRiver` and the affected formulas are `stateTransition` and `test1`. To determine the satisfiability of `test1`, we only need to evaluate the body of `stateTransition` and the body of `test1`. Moreover, if any unaffected formula is unsatisfiable, then we know the test is unsatisfiable even without invoking the evaluator. In practice, the technique improves the performance of ARepair because it does not involve any expensive SAT solving and is able to determine the test satisfiability with a minimal number of evaluator calls.

4.5 Hierarchical Caching

The evaluator-based approach to determine the test satisfiability can be further improved by our hierarchical caching algorithm. The idea is that we can reuse the previously evaluated result (i.e. valuation) of a formula if its subformulas evaluate to the same set of values as some subformulas we evaluated before. We explain hierarchical caching through the farmer example. Suppose we want to determine the satisfiability of `test1` (Figure 1b) by evaluating the fact formula `stateTransition`, and the created holes that correspond to nodes at level 7 of the most suspicious AST node, i.e. "`to`" and "`eats`" in Figure 2 highlighted in red. Also assume that hole $\bar{h}(\texttt{to})$ is first replaced by fragment "`none`" and hole $\bar{h}(\texttt{eats})$ is unchanged. We create a hierarchical cache for `test1` as follows. First, we invoke the evaluator for the fragments of both holes and find that "`none`" evaluates to $\varnothing$ and "`eats`" evaluates to {X0→C0,

Algorithm 1: ARepair algorithm.

```
Input: Faulty Alloy model M, test suite T.
Output: Fully fixed model or partially fixed model.
1  canFix = True
2  while canFix do
3      res = runTests(M)
4      if allTestsPassed(res) then return M // Full fix.
5      L = locateFaults(M, res)
6      if isEmpty(L) then return M // Partial fix.
7      canFix = False
8      if isFixed(M, L[0]) then
9          M = applyChange(M, L[0])
10         canFix = True
11     else
12         foreach n ∈ L do
13             patch = synthesize(M, n)
14             if isFixed(M, patch) then
15                 M = applyChange(M, patch)
16                 canFix = True
17                 break
18 return M // Partial fix.
```

C0→G0}. So we create mappings <"none", [∅]> for $\bar{h}$(`to`) and <"eats", [{X0→C0, C0→G0}]> for $\bar{h}$(`eats`). Since the join operator "·" in level 6 is the lowest common ancestor of both holes in level 7, a mapping <"∅. {X0→C0, C0→G0}", [∅]> is created for the join operator. Note that the key of the join operator is its string representation with all descendant holes replaced by their valuations under `test1`. The value of the mapping is obtained by evaluating the string representation of the join operator, i.e. "`none.eats`", which is ∅. We then create a mapping for the body of the declaring `crossRiver` predicate. But because the body has parameters ("`from`", "`from'`", "`to`", "`to'`"), we need to assign possible values to all parameters and create a mapping for the body <"$A_1A_2...A_n$", [{B_1},{B_2},...,{B_n}]>, where A_i means the string representation of the body (with the join operator "·" replaced with its actual valuation) given ith possible assignment of parameters, and B_i is the corresponding boolean result of the body formula in this case. We finally maps the cached value of `crossRiver`, i.e. [{B_1},{B_2},...,{B_n}], to the satisfiability of the `stateTransition` fact.

If the next fragment of hole $\bar{h}$(`to`) is "`item-Object`" which evaluates to ∅ ($\bar{h}$(`eats`) is unchanged), then we immediately know that `stateTransition` evaluates to the same result as when hole $\bar{h}$(`to`) is "`none`". Because the new keys we computed for other nodes, e.g. the join operator in level 6, are already in the cache. Therefore, we only invoke the evaluator once to evaluate "`item-Object`" instead of evaluating the big `stateTransition` body to determine its satisfiability. In general, the hierarchical cache reduces the input size of evaluator calls but increase the number of evaluator calls. In practice, we observe speed-ups for a majority of repairing problems and few repair problems suffer from a slow-down.

4.6 Repair Algorithm

Algorithm 1 shows the algorithm of ARepair. The algorithm takes as input a faulty Alloy model M and a test suite T that reveals the fault. The output is either a fully fixed model if all tests pass or a partially fixed model otherwise. In the worst scenario, ARepair is not able to fix any fault, in which case the partially fixed model is the original faulty model. Initially, we set *canFix* to *true* (line 1) and enter the loop (line 2). For each iteration in the loop, we first run all tests against M (line 3). If all tests pass, M is returned (line 4). Otherwise, we run AlloyFL to return a ranked list (L) of suspicious AST nodes (line 5). If L is empty, then the algorithm cannot fix the faulty model and it returns the latest state of M (line 6). Otherwise, we set *canFix* to *false* (line 7) and try to fix the faults. The algorithm checks if the most suspicious AST node (*L[0]*) has a potential fix (line 8). The *isFixed* check determines if we want to use the mutation or the synthesizer to fix the model. In general, the *isFixed* method returns true if the mutation makes X failing tests pass and Y passing tests fail, where $X > 0$ and $Y = 0$. In practice, X and Y can be arbitrary numbers as long as $X > Y$ holds, because we want to make sure the algorithm terminates. Since initially we have a finite number of failing tests and $X > Y$ makes sure that fewer tests are failing at each iteration. The total number of iterations is bounded by the number of initial failing tests. If *isFixed* (line 8) returns true, then we apply the mutation to M (line 9) and set *canFix* to *true* (line 10). Otherwise, we iterate over the ranked suspicious nodes (line 12) and try to fix the model using the synthesizer. For each suspicious node in L, we invoke the synthesizer to create holes, generate candidate fragments, explore the search spaces and find a potential patch (line 13). Then, the algorithm checks if the patch is a potential fix (line 14). The *isFixed* method in line 14 is similar to the method in line 8. If the patch is a fix, then we apply the patch to the model (line 15) with *canFix* set to *true* and exit the inner loop (line 12-17). Otherwise, we invoke the synthesizer on the next suspicious node in L. If the synthesizer cannot find a fix after exploring all suspicious nodes, the algorithm exits the outer loop (line 2) and returns the latest state of M.

If the resulting model passes all tests, ARepair simplifies the model to make it look more natural to the developer. For example, ARepair replaces "`to-none.eats`" with "`to`" because "`none.eats`" always evaluates to an empty set.

5 EVALUATION

We evaluate ARepair on 38 real faults collected from Alloy release 4.1, Amalgam [53] and Alloy homework solutions from graduate students. These faulty models contain various types of faults, i.e. overconstraints, underconstraints and a mixture of both. We define the number of faults as the number of incorrectly modeled Alloy paragraphs, e.g. signatures, predicates, functions and facts.

We address the following research questions in this section:

- **RQ1.** What is the repair efficacy of ARepair?
- **RQ2.** How does the quality of ARepair generated patches compared to human-written patches?
- **RQ3.** Why is ARepair unable to fix some models?

5.1 Experiment Setting

Unlike existing datasets that isolate faults for the repair techniques, e.g. Defects4J [25], we use the exact human-written faulty Alloy models as an input to ARepair. We use the Ochiai [1] formula

				all-combinations search strategy								base-choice search strategy							
Model	#Ast	#Test	#Flt	#Fix	Status	Types	SS	ES	FL(s)	EG(s)	EE(s)	#Fix	Status	Types	SS	ES	FL(s)	EG(s)	EE(s)
addr1	124	30	1	1	★	M	1	1	41.0	0.0	45.3	1	★	M	1	1	7.3	0.0	8.3
arr1	64	37	1	0	✗	–	1.7e5	1.3e5	36.0	13.5	291.4	1	★	S	4e4	88	1.8	1.9	6.7
arr2	80	37	1	1	★	M+S	3.4e7	5e6	11.6	1.9	3.4e3	1	★	M+S	3.4e7	178	11.4	1.9	16.9
bst1	186	124	1	1	✓	M	1	1	67.6	0.0	74.2	1	✓	M	1	1	46.4	0.0	48.8
bst2	161	124	3	–	∞	–	–	–	–	–	–	0	✗	–	2e17	1.6e5	678.6	597.2	4.8e3
bst3	165	124	2	–	∞	–	–	–	–	–	–	2	✓	M+S	2e8	4e3	241.3	1174	1.5e3
bempl1	51	25	1	0	✗	–	325	325	2.6	0.4	5.3	0	✗	–	325	67	2.6	0.3	5.3
cd1	59	31	2	1	†	M	1e4	993	5.9	1.3	10.6	2	✓	M+S	1.9e5	688	6.0	3.9	12.7
cd2	50	31	1	1	✓	S	966	350	2.3	0.6	6.4	1	✓	S	2.4e5	810	2.2	4.7	11.2
ctree1	71	22	1	1	✓	M	1	1	5.6	0.0	6.5	1	✓	M	1	1	5.5	0.0	6.4
dll1	109	50	2	2	✓	S	4.9e4	8522	19.5	2.3	42.1	2	✓	S	6.7e4	239	19.0	8.3	34.1
dll2	105	50	2	2	✓	M+S	4.9e4	8521	26.7	1.7	46.9	2	✓	M+S	6.6e4	192	27.3	7.7	39.8
dll3	101	50	3	–	∞	–	–	–	–	–	–	0	✗	–	1.7e9	2.1e4	31.3	30.4	94.7
dll4	109	50	1	1	✓	M+S	4.9e4	8384	16.3	1.7	36.2	1	✓	M+S	6.6e4	191	16.5	7.7	28.8
farmer1	180	76	1	–	∞	–	–	–	–	–	–	1	✓	M+S	7.6e13	4556	140.7	1.6e4	4.5e4
fsm1	116	15	2	2	★	M	2	2	7.0	0.0	7.7	2	★	M	2	2	6.9	0.0	7.7
fsm2	93	15	1	1	★	M	1	1	3.8	0.0	4.7	1	★	M	1	1	3.9	0.0	4.7
grade1	71	42	1	–	∞	–	–	–	–	–	–	1	✓	S	1.5e9	1e3	5.2	9.5	739.6
other1	68	22	1	1	★	S	7593	586	2.4	0.7	8.3	0	✗	–	1.7e4	387	2.4	0.7	8.4
stu1	213	98	1	1	✓	S	9.3e4	836	52.6	6.7	85.1	1	✓	S	1.7e6	186	26.3	1.8	46.6
stu2	195	98	2	–	∞	–	–	–	–	–	–	1	✗	S	9.6e12	9905	141.7	40.0	453.5
stu3	237	98	2	–	∞	–	–	–	–	–	–	0	✗	–	1.7e15	3.5e4	347.9	355.9	2.8e3
stu4	190	98	1	1	✓	S	9.3e4	836	53.6	6.6	85.6	1	✓	S	1.7e6	186	26.8	1.9	46.9
stu5	235	98	1	1	✓	S	9.3e4	836	55.0	6.7	87.9	1	✓	S	1.7e6	186	27.5	1.9	48.4
stu6	191	98	3	–	∞	–	–	–	–	–	–	2	✗	M+S	3.8e8	4e3	120.2	22.4	282.4
stu7	174	98	2	2	✓	M+S	5.4e5	1e4	188.9	15.4	265.7	1	✗	S	9.3e10	2.5e4	213.7	291.6	1.7e3
stu8	213	98	1	1	✓	S	1.4e4	1e4	33.4	7.9	78.7	1	✓	S	1.4e4	120	33.4	7.5	54.7
stu9	198	98	1	1	★	M	1	1	49.0	0.0	51.0	1	★	M	1	1	49.9	0.0	51.9
stu10	200	98	1	1	✓	S	9.3e4	836	63.7	6.4	95.9	1	✓	S	1.7e6	186	30.4	1.9	50.5
stu11	221	98	1	1	✓	S	1.4e7	4317	97.6	20.7	174.5	1	✓	S	1.4e7	571	65.3	16.4	131.5
stu12	201	98	2	2	✓	M+S	9.3e4	887	179.8	8.1	224.6	2	✓	M+S	1.7e6	264	152.9	3.5	186.9
stu13	221	98	1	1	★	M	1	1	64.3	0.0	66.4	1	★	M	1	1	64.5	0.0	66.5
stu14	183	98	3	–	∞	–	–	–	–	–	–	2	✗	M+S	3.8e8	4e3	105.5	23.4	266.2
stu15	207	98	1	1	✓	S	9.3e4	836	68.6	6.8	100.7	1	✓	S	1.7e6	186	33.7	2.0	53.6
stu16	113	98	4	–	∞	–	–	–	–	–	–	0	✗	–	5.9e5	6901	43.0	67.9	250.3
stu17	190	98	2	–	∞	–	–	–	–	–	–	1	✗	S	3.5e8	3741	61.0	22.5	205.5
stu18	207	98	3	3	✓	M+S	2.9e9	3.7e5	160.6	6.8	1.1e3	3	✓	M+S	2.9e9	409	115.4	7.4	152.1
stu19	216	98	2	–	∞	–	–	–	–	–	–	1	✗	S	1e13	8221	194.1	9e3	9596

Figure 9: ARepair Results. Times are in seconds. – denotes not applicable.

to rank suspicious AST nodes in AlloyFL, because existing studies [68, 85] show that Ochiai is effective. The expression generator generates different sizes of expressions based on the level of the holes in the suspicious AST. We set the expression size to 3 for the deepest level of holes in each suspicious AST. The expression size increases by 1 for holes at depth D_{i-1} compared to holes at depth D_i where $D_i - D_{i-1} = 1$, up to a maximum expression size of 6. For the all-combinations search strategy, we partition the candidate fragments for each hole into 10 parts (i.e. $k_i = 10$ in Figure 7), and we set the maximum number of combinations of candidate fragments to explore to 10000 (per level of holes). For the base-choice search strategy, we set the maximum number of candidate fragments to explore for each hole to 1000. The AUnit tests we use to validate the patches are *automatically generated* using MuAlloy [74] so that they are able to detect all non-equivalent mutant models [71]. Additionally, the authors manually inspect the generated tests and add some new tests to cover different corner cases. The manually written tests account for <7% of the total tests.

We validate the correctness of a generated patch by both inspecting them manually and using the Alloy analyzer to perform a scope bounded equivalence check. The human-written patches are written with the intention of introducing small perturbations that are sufficient to fix the faults. We terminate ARepair once it finds a patch that passes all tests.

All experiments are performed on Ubuntu 16.04 LTS with 2.4GHz Intel Xeon CPU and 16 GB memory. To save space, we denote the all-combinations search strategy as *AC* and the base-choice search strategy as *BC* in the following sections.

5.2 Repair Efficacy

Figure 9 shows the detailed results for ARepair. *Model*, *#Ast*, *#Test* and *#Flt* show the name, the number of AST nodes, the number of tests and the number of faults, respectively, for each subject model. **farmer1** is from Alloy release 4.1. **addr1**, **bempl1**, **grade1** and **other1** are from Amalgam [53]. The rest models are from graduate student solutions. Student solutions for the same question share the same test suite. *#Fix* shows the number of faults a search strategy is able to fix. *Status* shows the repair status. ★ means the generated patch is syntactically identical to the human-written patch. ✓ means the generated patch is syntactically different but semantically equivalent to the human-written path. † means the patch is plausible (incorrect but passes all tests). ✗ means ARepair fails to generate a patch that pass all tests. ∞ means the repair times out after 15 hours. *Types* shows whether the fix requires mutations (M) or synthesis (S). *SS* shows the search space size, which is defined as the sum of the number of combinations of candidate fragments (including applied mutations for fixes) to consider in each iteration. *ES* is the actual number of combinations (or mutations) ARepair tried. *FL* is the fault localization time. *EG* is the expression generation time. *EE* is the end-to-end time. All times are in seconds.

The entire experiments contain 38 faulty models and 62 individual faults. AC is able to fix 24 models and 31 faults. BC is able to fix 26 models and 42 faults. Additionally, AC times out ($\geq$ 15h) for 12 models while BC finishes all models in 15h. AC is able to fix 2 models that BC is not able to fix, e.g. **other1** and **stu7**. BC is able to fix 5 models that AC is not able to fix (including 1 plausible patch for **cd1**), e.g. **arr1**, **bst3**, **cd1**, **farmer1** and **grade1**. Many models require both mutations and synthesis for a complete fix, e.g. **arr2** and **dll2**. AC's search space ranges from 1 to 2.9e9 and the maximum size of the explored space is 5e6. BC's search space ranges from 1 to 2e17 and the maximum size of the explored space is 1.6e5. We can see that BC explores less of its search space than AC, though BC typically has a much larger search space. In general, BC runs faster than AC, with the exceptions of **cd1**, **cd2** and **stu7**. For AC, the fault localization time ranges from 2.3 sec to 188.9 sec and the maximum expression generation time is 20.7 sec, excluding timeout cases. For BC, the fault localization time ranges from 1.8 sec to 678.6 sec and the maximum expression generation time is 1.6e4 sec. Typically, AC times out for models whose expression generation time is large ($\geq$ 1000s) under BC. A large expression generation time is a reflection of ARepair producing a large number of expressions, resulting in large search spaces. This means that when there are so many combinations of candidate fragments to consider, AC typically times out. In comparison, despite the large number of expressions produced, BC explores much less space and thus is faster. However, BC can run into its local optimum. For example, the explored space for **other1** is less than 600 for both BC and AC, but BC cannot fix the model. Overall, AC and BC are complementary and BC is superior in the sense that it takes less time to run and fixes more faults.

5.3 Patch Quality

To answer **RQ2**, we find that BC generates 26 patches that pass all tests (all patches are correct and 7 patches exactly match human-written patches). AC generates 24 patches that pass all tests (23 patches are correct; 7 patches exactly match human-written patches; and 1 patch is plausible but incorrect). We compare the generated patches that are syntactically different but semantically equivalent to human-written patches. In addition to patches for the faulty farmer model (Figure 3), Figure 10 compares ARepair generated patches and human-written patches for **bst1** (A and B), **cd2** (C and D) and **stu8** (E and F). The `Sorted` predicate in **bst1** models that the value of the current node should be greater than values of its left descendants and less than values of its right descendants. The developer incorrectly use "`n.^left`" to represent the domain of `n`'s left descendants. The correct domain should be "`n.left.*(left+right)`" as shown in the human-written patch. The generated patch restricts the domain to be "`n.^left.*right`" which means all nodes that can be reachable from `n` by first following one or more `left` relation and then zero or more `right` relation. The `Acyclic` predicate in **cd2** models that a class does not transitively extend itself. The faulty model does not consider the transitivity requirement, which is fixed in the human-written patch by replacing "`c = c.ext`" with "`c in c.^ext`". The generated patch uses "`c = c & c.^ext`" which

(A) Human-written patch for bst1.

```
1.   pred Sorted() { all n: Node {
2.-    all n2: n.^left | n2.elem < n.elem
3.+    all n2: n.left.*(left+right) | n2.elem < n.elem
4.-    all n2: n.^right | n2.elem > n.elem}}
5.+    all n2: n.right.*(left+right) | n2.elem > n.elem}}
```

(B) ARepair generated patch for bst1.

```
1.   pred Sorted() { all n: Node {
2.-    all n2: n.^left | n2.elem < n.elem
3.+    all n2: n.^left.*right | n2.elem < n.elem
4.-    all n2: n.^right | n2.elem > n.elem}}
5.+    all n2: n.^right.*left | n2.elem > n.elem}}
```

(C) Human-written patch for cd2.

```
1.   pred Acyclic() {
2.-    no c: Class | c = c.ext }
3.+    no c: Class | c in c.^ext }
```

(D) ARepair generated patch for cd2.

```
1.   pred Acyclic() {
2.-    no c: Class | c = c.ext }
3.+    no c: Class | c = c & c.^ext }
```

(E) Human-written patch for stu8.

```
1.   pred Sorted(This: List) {
2.-    all n: Node | n.elem<=n.link.elem }
3.+    all n: Node | some n.link => n.elem<=n.link.elem }
```

(F) ARepair generated patch for stu8.

```
1.   pred Sorted(This: List) {
2.-    all n: Node | n.elem<=n.link.elem }
3.+    all n: link.Node | n.elem<=n.link.elem }
```

Figure 10: Comparison of ARepair generated patches and human-written patches.

states that no class is equal to the intersection of the class and all its subclasses, transitively. The `Sorted` predicate in **stu8** models a linked list sorted in descending order of the node values. The faulty model does not allow the existence of any list with a single node (without any `link`). The human-written patch allows such cases by stating that if a node `n` has a subsequent node following the link, then its value should be less than or equal to the value of its subsequent node. The generated patch instead modifies the domain to restrict the less than or equal relation only applying to nodes that have a subsequent node.

The authors check correct patches that are syntactically different from human-written patches and find that these patches are easy to understand in general. There are rare cases that ARepair generates some complex expressions that can be further simplified through semantic reasoning. Additionally, ARepair generates a patch which fixes a fact instead of the predicate the developer would fix for **ctree1**.

5.4 Limitation

To answer **RQ3**, we manually inspect all faulty models that ARepair is unable to fix. The reasons are categorized as follow:

(1) The repair requires synthesizing predicate and function calls. For example, one of the property to fix in **bst2** requires invoking predicates and functions.
(2) The repair requires moving a field declaration from one signature to another, e.g. **bempl1**.
(3) The repair requires creation of new syntactic structures. For example, **dll3** models a property using a single quantifier, but the model needs two. **stu2** has a formula with the structure $(\alpha \Rightarrow \beta) \mid\mid \gamma$, but the correct fix requires $\alpha \Rightarrow \beta$ *else* γ, where α, β and γ are formulas. **stu6** is overconstrained and the fix requires

creating a disjunction of a new formula and an existing formula. **dll3** and **stu16** have empty predicates and require ARepair to synthesize formulas from scratch.

(4) Both AC and BC search strategies are greedy and may run into a local optimum. For example, a correct patch of **other1** requires changing two formulas at the same time and BC runs into a local optimum that leads to a repair failure. Similarly, AC runs into a local optimum for **arr1**.

We find that the majority of the faults ARepair is unable to fix fall under category 3, followed by category 4. To handle faults in category 3, we can add repair templates that introduce new syntactic structures if the current version of ARepair is not able to find a correct patch. New search strategies can be designed to address faults under category 4. From our experiment, ARepair is able to handle a majority of the faulty models (28 out of 38) and we plan to handle the limitations in future works.

6 THREATS TO VALIDITY

There exists several threats to the validity of our results. Many of the parameters in the implementation and experimental setup were chosen by heuristics. They may not represent the optimum set of parameter values. Moreover, these parameter values may not generalize to other unseen faulty models. ARepair's ability to fix faulty models depends on the fault localization technique and the AUnit test suite. Our experiment results may vary if we use a different fault localization technique, e.g. Tarantula [23]. If the test suite is too weak to capture the desired model properties, ARepair may give too many plausible but incorrect patches. The real faulty models we use in the experiment are limited in the sense that most of them are written by graduate students. So the experiment results may not generalize to faulty models written by experienced developers. However, we collected our set of subject faulty models to the best of our ability.

The AUnit tests (e.g., the test in Figure 1b), when written manually, can require some effort. In this paper, a majority of the tests (> 93%) are generated by MuAlloy [74] in which case the manual effort is substantially reduced. In general, the manual effort can be reduced by writing *partial* tests that provide valuations for a subset of the relations declared in the model. For example, the test in Figure 1b can omit the constraints on various relations (e.g., `State`, `near`, and `far`) because these constraints are irrelevant to the property the test is intended to check.

7 RELATED WORK

ARepair is a *generate-and-validate* repair technique for declarative models written in Alloy. The technique is able to fix models with multiple faults or faults that require fixes at multiple locations. ARepair does not have any repair templates, instead it creates holes in suspicious AST nodes level by level and can fix different kinds of bugs. The idea of combining mutations (from MBFL) and a synthesizer to repair faulty programs is new. The base-choice search strategy reduces the exploration space and is different from search strategies of existing repair techniques. The hierarchical caching reduces the input sizes of evaluator calls and is different from existing memoization techniques. Next, we highlight the main areas of work related to ARepair.

Generate-and-Validate Repair. The *generate-and-validate* repair techniques apply a set of code transformations to generate program candidates and validate each candidate under the given test suite. These techniques implement different search strategies, e.g. genetic algorithms [81], semantic search [27], random search [59] and adaptive search [80], to explore the immense search space of repair candidates. Researchers also proposed other repair techniques that remove program functionalities [60], create program variants [8, 11], leverage dynamic program state [18, 19, 86], or focus on improving performance by removing bottlenecks in concurrent programs [87]. *Astor* [43] is a repair library that implements existing techniques to fix Java code. Techniques that prioritize patches are built based on human-written code [29, 37, 66, 83], historical data [14, 34], document analysis [36, 62, 84], anti-patterns [72] and test generation [82].

Constraint-Solving Repair. The *constraint-solving* repair techniques use the semantics of the faulty program and translate the repair problem into a constraint solving problem. Then, the constraint solving problem is solved by an off-the-shelf solver to find a repair that satisfies all inferred specifications. The constraints can be inferred from test executions [12, 38, 64] or semantic analysis [10, 26, 33, 55]. Other techniques use formal specifications [17, 30, 67, 79] or infer invariants [13, 24, 58, 63] to fix programs.

Declarative Debugging. The fundamental idea of declarative debugging is that the programmer (or some oracle) has an intended interpretation of the program and debuggers can query the programmer to obtain this information. The debugger compares the intended interpretation of a (buggy) program with its (incorrect) actual behavior on some computation. The cause of the difference is isolated to a small section of code which must contain a bug. Declarative debugging was first introduced in Prolog [65] and then extended for functional and logic programs [50, 51, 57]. Researchers also developed program repair technique for SQL [5, 16].

Alloy. Over the past years, many extensions have been built for Alloy [9, 15, 45, 46, 70, 75]. Aluminum [54] generates minimal instances to make it easy for users to inspect. Amalgam [53] allows users to ask why and why not a relation exist in an Alloy instance. ASketch [76, 78] provides a sketching framework for Alloy.

8 CONCLUSION

This paper introduces a *generate-and-validate* repair technique, ARepair, to fix faulty Alloy models. ARepair leverages a mutation-based fault localization technique, an expression generator and a synthesizer to repair various kinds of faults. ARepair is enumeration-based and it enbodies two search strategies, i.e. the all-combination strategy and the base-choice strategy. ARepair implements various optimizations, including the use of modulo test input pruning to remove equivalent expressions, the construction of dependency graph to reduce evaluator calls, and the employment of a hierarchical cache to reduce evaluator input size. The experimental results show that ARepair works well in fixing real faulty models.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments. This work was partially supported by the US National Science Foundation under Grants Nos. CCF-1718903.

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2007. On the Accuracy of Spectrum-based Fault Localization. In *TAICPART-MUTATION*.

[2] Christoffer Quist Adamsen, Anders Møller, Rezwana Karim, Manu Sridharan, Frank Tip, and Koushik Sen. 2017. Repairing event race errors by controlling nondeterminism. In *ICSE*.

[3] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling enumerative program synthesis via divide and conquer. In *TACAS*.

[4] Paul Ammann and Jeff Offutt. 2008. *Introduction to Software Testing*.

[5] Marcelo Arenas, Leopoldo Bertossi, and Jan Chomicki. 2001. Specifying and querying database repairs using logic programs with exceptions. In *Flexible Query Answering Systems*.

[6] Hamid Bagheri and Sam Malek. 2016. Titanium: Efficient Analysis of Evolving Alloy Specifications. In *FSE*.

[7] Yan Cai, Lingwei Cao, and Jing Zhao. 2017. Adaptively Generating High Quality Fixes for Atomicity Violations. In *FSE*.

[8] Antonio Carzaniga, Alessandra Gorla, Andrea Mattavelli, Nicolò Perino, and Mauro Pezzè. 2013. Automatic Recovery from Runtime Failures. In *ICSE*.

[9] Alcino Cunha and Nuno Macedo. 2018. Validating the Hybrid ERTMS/ETCS Level 3 Concept with Electrum. In *ABZ*.

[10] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. 2016. Qlose: Program Repair with Quantitative Objectives. In *CAV*.

[11] Vidroha Debroy and W. Eric Wong. 2010. Using Mutation to Automatically Suggest Fixes for Faulty Programs. In *ICST*.

[12] Favio Demarco, Jifeng Xuan, Daniel Le Berre, and Martin Monperrus. 2014. Automatic repair of buggy if conditions and missing preconditions with SMT. In *CSTVA*.

[13] Brian Demsky, Michael D. Ernst, Philip J. Guo, Stephen McCamant, Jeff H. Perkins, and Martin C. Rinard. 2006. Inference and enforcement of data structure consistency specifications. In *ISSTA*.

[14] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *POPL*.

[15] MR Frias, Juan P Galeotti, Carlos G López Pombo, and Nazareno M Aguirre. 2005. DynAlloy: upgrading alloy with actions. In *ICSE*.

[16] Divya Gopinath, Sarfraz Khurshid, Diptikalyan Saha, and Satish Chandra. 2014. Data-guided repair of selection statements. In *ICSE*.

[17] Divya Gopinath, Muhammad Zubair Malik, and Sarfraz Khurshid. 2011. Specification-Based Program Repair Using SAT. In *TACAS*.

[18] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. 2018. SketchFix: A Tool for Automated Program Repair Approach Using Lazy Candidate Generation. In *FSE*.

[19] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. 2018. Towards Practical Program Repair with On-Demand Candidate Generation. In *ICSE*.

[20] Changyun Huang, Yasutaka Kamei, Kazuhiro Yamashita, and Naoyasu Ubayashi. 2013. Using Alloy to Support Feature-based DSL Construction for Mining Software Repositories. In *SPLC*.

[21] Si Huang, Myra B. Cohen, and Atif M. Memon. 2010. Repairing GUI Test Suites Using a Genetic Algorithm. In *ICST*.

[22] Daniel Jackson. 2002. Alloy: A Lightweight Object Modelling Notation. *TOSEM* (2002).

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

[24] Frolin S. Ocariza Jr., Karthik Pattabiraman, and Ali Mesbah. 2014. Vejovis: suggesting fixes for JavaScript faults. In *ICSE*.

[25] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *ISSTA*.

[26] Shalini Kaleeswaran, Varun Tulsian, Aditya Kanade, and Alessandro Orso. 2014. MintHint: automated synthesis of repair hints. In *ICSE*.

[27] Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. 2015. Repairing Programs with Semantic Code Search. In *ASE*.

[28] Sarfraz Khurshid, Darko Marinov, Ilya Shlyakhter, and Daniel Jackson. 2003. A Case for Efficient Solution Enumeration. In *SAT*.

[29] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *ICSE*.

[30] Etienne Kneuss, Manos Koukoutos, and Viktor Kuncak. 2015. Deductive Program Repair. In *CAV*.

[31] Yoann Laurent, Reda Bendraou, Souheib Baarir, and Marie-Pierre Gervais. 2014. Alloy4SPV: A Formal Framework for Software Process Verification. In *ECMFA*.

[32] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *PLDI*.

[33] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: syntax- and semantic-guided repair synthesis via programming by examples. In *FSE*.

[34] Xuan-Bach D. Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *SANER*.

[35] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *ICSE*.

[36] Chen Liu, Jinqiu Yang, Lin Tan, and Munawar Hafiz. 2013. R2Fix: Automatically Generating Bug Fixes from Bug Reports. In *ICST*.

[37] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic inference of code transforms for patch generation. In *FSE*.

[38] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *FSE*.

[39] Sonal Mahajan, Abdulmajeed Alameer, Phil McMinn, and William G. J. Halfond. 2017. Automated repair of layout cross browser issues using search-based techniques. In *ISSTA*.

[40] Ferney A. Maldonado-Lopez, Jaime Chavarriaga, and Yezid Donoso. 2014. Detecting Network Policy Conflicts Using Alloy. In *ABZ*.

[41] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. 2011. CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In *MODELS*.

[42] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. 2011. CDDiff: Semantic Differencing for Class Diagrams. In *ECOOP*.

[43] Matias Martinez and Martin Monperrus. 2016. ASTOR: a program repair library for Java (demo). In *ISSTA*.

[44] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. DirectFix: Looking for Simple Program Repairs. In *ICSE*.

[45] Aleksandar Milicevic, Joseph P. Near, Eunsuk Kang, and Daniel Jackson. 2015. Alloy*: A General-purpose Higher-order Relational Constraint Solver. In *ICSE*.

[46] Facundo Molina, César Cornejo, Renzo Degiovanni, Germán Regis, Pablo F Castro, Nazareno Aguirre, and Marcelo F Frias. 2016. An Evolutionary Approach to Translate Operational Specifications into Declarative Specifications. In *BSFM*.

[47] Vajih Montaghami and Derek Rayside. 2012. Extending Alloy with Partial Instances. In *ABZ*.

[48] S. Moon, Y. Kim, M. Kim, and S. Yoo. 2014. Ask the Mutants: Mutating Faulty Programs for Fault Localization. In *ICST*.

[49] Mariano M. Moscato, Carlos G. Lopez Pombo, and Marcelo F. Frias. 2014. Dynamite: A Tool for the Verification of Alloy Models Based on PVS. *ACM Trans. Softw. Eng. Methodol.* (2014).

[50] Lee Naish. 1997. A Declarative Debugging Scheme. *Journal of Functional and Logic Programming* (1997).

[51] Lee Naish. 2000. A three-valued declarative debugging scheme. In *Computer Science Conference*.

[52] Timothy Nelson, Christopher Barratt, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. 2010. The Margrave Tool for Firewall Analysis. In *LISA*.

[53] Tim Nelson, Natasha Danas, Daniel J. Dougherty, and Shriram Krishnamurthi. 2017. The Power of "Why" and "Why Not": Enriching Scenario Exploration with Provenance. In *FSE*.

[54] Tim Nelson, Salman Saghafi, Daniel J Dougherty, Kathi Fisler, and Shriram Krishnamurthi. 2013. Aluminum: principled scenario exploration through minimality. In *ICSE*.

[55] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *ICSE*.

[56] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: Mutation-based Fault Localization. *STVR* (2015).

[57] Luís Moniz Pereira. 1986. Rational debugging in logic programming. In *ICLP*.

[58] Jeff H. Perkins, Sunghun Kim, Samuel Larsen, Saman P. Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin C. Rinard. 2009. Automatically patching errors in deployed software. In *SOSP*.

[59] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The strength of random search on automated program repair. In *ICSE*.

[60] Zichao Qi, Fan Long, Sara Achour, and Martin C. Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *ISSTA*.

[61] Natali Ruchansky and Davide Proserpio. 2013. A (Not) NICE Way to Verify the Openflow Switch Specification: Formal Modelling of the Openflow Switch Using Alloy. *SIGCOMM* (2013).

[62] Ripon K Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R Prasad. 2017. ELIXIR: effective object oriented program repair. In *ASE*.

[63] Hesam Samimi, Max Schäfer, Shay Artzi, Todd D. Millstein, Frank Tip, and Laurie J. Hendren. 2012. Automated repair of HTML generation errors in PHP applications using string constraint solving. In *ICSE*.

[64] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *ICSE*.

[65] Ehud Y. Shapiro. 1983. *Algorithmic Program DeBugging*. MIT Press.

[66] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. 2015. Automatic error elimination by horizontal code transfer across multiple applications. In *PLDI*.

[67] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. In *PLDI*.

[68] Friedrich Steimann, Marcus Frenkel, and Rui Abreu. 2013. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In *ISSTA*.

[69] Allison Sullivan, Kaiyuan Wang, and Sarfraz Khurshid. 2018. AUnit: A Test Automation Tool for Alloy. In *ICST*.

[70] Allison Sullivan, Kaiyuan Wang, Sarfraz Khurshid, and Darko Marinov. 2017. Evaluating State Modeling Techniques in Alloy. In *SQAMIA*.

[71] Allison Sullivan, Kaiyuan Wang, Razieh Nokhbeh Zaeem, and Sarfraz Khurshid. 2017. Automated Test Generation and Mutation Testing for Alloy. In *ICST*.

[72] Shin Hwei Tan, Hiroaki Yoshida, Mukul R. Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in search-based program repair. In *FSE*.

[73] Emina Torlak and Daniel Jackson. 2007. Kodkod: A Relational Model Finder. In *TACAS*.

[74] Kaiyuan Wang, Allison Sullivan, and Sarfraz Khurshid. 2018. MuAlloy: A Mutation Testing Framework for Alloy. In *ICSE*.

[75] Kaiyuan Wang, Allison Sullivan, Manos Koukoutos, Darko Marinov, and Sarfraz Khurshid. 2018. Systematic Generation of Non-Equivalent Expressions for Relational Algebra. In *ABZ*.

[76] Kaiyuan Wang, Allison Sullivan, Darko Marinov, and Sarfraz Khurshid. 2018. ASketch: A Sketching Framework for Alloy. In *FSE*.

[77] Kaiyuan Wang, Allison Sullivan, Darko Marinov, and Sarfraz Khurshid. 2018. Fault Localization for Declarative Models in Alloy. In *eprint arXiv:1807.08707*.

[78] Kaiyuan Wang, Allison Sullivan, Darko Marinov, and Sarfraz Khurshid. 2018. Solver-based Sketching Alloy Models using Test Valuations. In *ABZ*.

[79] Yi Wei, Yu Pei, Carlo A. Furia, Lucas Serpa Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. 2010. Automated fixing of programs with contracts. In *ISSTA*.

[80] Westley Weimer, Zachary P. Fry, and Stephanie Forrest. 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In *ASE*.

[81] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *ICSE*.

[82] Qi Xin and Steven P. Reiss. 2017. Identifying test-suite-overfitted patches through test case generation. In *ISSTA*.

[83] Qi Xin and Steven P Reiss. 2017. Leveraging syntax-related code for automated program repair. In *ASE*.

[84] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *ICSE*.

[85] Jifeng Xuan and Martin Monperrus. 2014. Learning to Combine Multiple Ranking Metrics for Fault Localization. In *ICSME*.

[86] Zijiang Yang, Jinru Hua, Kaiyuan Wang, and Sarfraz Khurshid. 2018. EdSynth: Synthesizing API Sequences with Conditionals and Loops. In *ICST*.

[87] Tingting Yu and Michael Pradel. 2017. Pinpointing and repairing performance bottlenecks in concurrent programs. *ESE* (2017).