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.

CSC CONCEPTS
• Software and its engineering \(\rightarrow\) Error handling and recovery;

KEYWORDS
Model repair, first-order logic, ARepair

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].

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.

∗Now at North Carolina A&T State University.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASE ’18, September 3–7, 2018, Montpellier, France
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5935-5/18/09...$15.00
https://doi.org/10.1145/3238147.3238162
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 thevaluations 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 : set Object )
5. fact eating ( eats = Fox->Chiken + 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.

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

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 }
20. // More tests ...

Figure 1: Faulty Farmer Example and Tests.

Figure 2: First Suspicious Node for Faulty Farmer Example.

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 "&&" to "|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". BOD deletes an operand of a logical operator, e.g. "a |b| to "b. BDE 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
we describe the search strategies (Section ASE ’18, September 3–7, 2018, Montpellier, France Kaiyuan Wang, Allison Sullivan, and Sarfraz Khurshid).

The intuition is that AlloyFL is designed to mutate upper-level operators (root level) until some failing test passes and no passing test fails. Thus, the command expected to be unsatisfiable. In this paper, each command invokes an AUnit 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 
\[ \sqrt{\text{total failed} \times (\text{failed}(e) + \text{passed}(e))} \]
where \( \text{failed}(e) \) and \( \text{passed}(e) \) are the number of tests that failed and passed (with respect to the original faulty model) that kill the mutant \( e \), and \( \text{total failed} \) 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.

\[ 0 \] means that the command to be unsatisfiable. In this paper, each command invokes an AUnit 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 
\[ \sqrt{\text{total failed} \times (\text{failed}(e) + \text{passed}(e))} \]
where \( \text{failed}(e) \) and \( \text{passed}(e) \) are the number of tests that failed and passed (with respect to the original faulty model) that kill the mutant \( e \), and \( \text{total failed} \) 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 operators 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 \( \phi h \)), logical operator holes (denoted by \( \phi oh \)), comparison operator holes (denoted by \( \phi oh \)), implication holes (denoted by \( \phi i h \)), cardinality holes (denoted by \( \phi rh \)), boolean holes (denoted by \( \phi bh \)) and expression holes (denoted by \( \phi eh \)). The value of \( \phi h \) can be one of \( \text{"all"}, \text{"no"}, \text{"some"}, \text{"one"} \) or \( \text{"one"} \). The value of \( \phi oh \) can be either \( \text{"&"} \) or \( \text{"\|"} \). The value of \( \phi coh \) can be one of \( \text{"="}, \text{"\langle\text{<\"}} \), \( \text{"\langle\text{>\"}} \) or \( \text{"\langle\text{\le\"}} \). The value of \( \phi rh \) can be one of \( \text{"no"}, \text{"one"}, \text{"one"} \) or \( \text{"one"} \). The value of \( \phi bh \) can be either empty \( \text{""} \) or \( \text{"!"} \). The value of \( \phi eh \) 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 \( \text{"&"} \) as shown in Figure 2). To create holes for the root node, we can apply the schema for the conjunction node. States that the holes we should create include a logical operator hole \( \phi oh \), 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 (\( \text{"="} \)), 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.
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 nodes 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 $10^{6}$. 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 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 ($P_{i1}^n$ to $P_{ik_i}^n$) of equal size. Next, we create size-$n$ tuples $U = \{ (x_1, x_2, \ldots, x_n) | \land_{i=1}^n x_i \in [1,k_i] \}$ and

Figure 5: Hole creation schemas for Alloy Surface Syntax. $\hbar(x)$ computes the holes for syntax $x$. $\alpha$, $\beta$, $\gamma$ denote formulas which evaluate to true or false. $\phi$ and $\psi$ denote expressions which evaluate to relations. $i : \phi$ denote tuple membership $i \in \phi$.

Figure 6: Expression generation syntax.

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

```plaintext
expression e := uop [ e ] | e bop [ e, e ] | atom \_ [ e ] [ const ]
unary op uop ::= * | + | - | ~ | .
binary op bop ::= * | & | - | .
atomic exp atom ::= sig \_ [ e ] | field \_ [ e ] | param \_ [ e ] | var \_ [ e ]
constant const ::= none | iden | uniq | Int | 0 | 1
```

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 $t_1$ and $t_2$. 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 val-

<table>
<thead>
<tr>
<th>Meaning</th>
<th>Schema</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cartesian product</td>
<td>$h(\phi \times \psi) \Rightarrow eh$</td>
</tr>
<tr>
<td>Union</td>
<td>$h(\phi \cup \psi) \Rightarrow eh$</td>
</tr>
<tr>
<td>Set difference</td>
<td>$h(\phi \setminus \psi) \Rightarrow eh$</td>
</tr>
<tr>
<td>Domain restriction</td>
<td>$h(\phi \langle \psi \rangle \Rightarrow eh$</td>
</tr>
<tr>
<td>Transitive closure</td>
<td>$h(\psi) \Rightarrow eh$</td>
</tr>
<tr>
<td>Inverse relational join</td>
<td>$h(\psi(\phi)) \Rightarrow eh$</td>
</tr>
<tr>
<td>Cardinality</td>
<td>$h(n) \Rightarrow eh$</td>
</tr>
<tr>
<td>Identity relation (binary)</td>
<td>$h(iden) \Rightarrow eh$</td>
</tr>
<tr>
<td>Conjunction</td>
<td>$h(\alpha \land \beta) \Rightarrow h(\alpha) \land h(\beta)$</td>
</tr>
<tr>
<td>Implication</td>
<td>$h(\alpha \Rightarrow \beta) \Rightarrow h(\alpha) \Rightarrow h(\beta)$</td>
</tr>
<tr>
<td>If-then-else</td>
<td>$h(\alpha \beta : \gamma) \Rightarrow h(\alpha) \beta h(\gamma)$</td>
</tr>
<tr>
<td>Relational equality</td>
<td>$h(\phi = \psi) \Rightarrow h(\phi) h(\psi)$</td>
</tr>
<tr>
<td>Universal quantification</td>
<td>$h(\forall t : \phi(x(t))) \Rightarrow \forall h(\phi) h(x(t))$</td>
</tr>
</tbody>
</table>

For test $t_1$, $n$ can only be $n=0$ and $link=\emptyset$. It is easy to see that “n”, “link” and “link(Node)n” evaluate to the same value [no] and are thus equivalent with respect to $t_1$. For test $t_2$, $n$ can be $n=0$ or $n=1$, and $link$ is in $[\emptyset \rightarrow N_0, N_1 \rightarrow N_0]$. “n”, “link”, “Node-n” and “link(Node)n” are equivalent with respect to $t_2$ both when $n=0$ and $n=1$. 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,
sort the tuples first by $\sum_{i=1}^{n} x_i$ and then by $\max_{i \in [1, n]} x_i - \min_{i \in [1, n]} x_i$ to get 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 $C = \{(f_1, \ldots, f_n) \mid \land_{i=1}^{n} f_i \in P^x_i\}$. 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 $hole_1$ with the candidate fragments of the rest holes unchanged. For each $hole_1$, 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 $hole_1$ 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 $Object_{mult}$) 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 $eats_{mult}$). "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 $Farmer_{mult}$). 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.

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 $h(to)$ is first replaced by fragment "none" and hole $h(eats)$ 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 ∅ and "eats" evaluates to $\{X0\rightarrow C0,$...}.
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     end
19 end
20 return M // Partial fix.

C0→G0]. So we create mappings <<“none”, ∅>> for h(to) and <<“eats”, [{X0→C0, C0→G0}]>> for 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 <<“A1.A2...A_n”, [{B1}, {B2}, ..., {B_n}]>>, where A_i means the string representation of the body (with the join operator “.” replaced with its actual valuation) given i-th 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. [{B1}, {B2}, ..., {B_n}], to the satisfiability of the stateTransition fact.

If the next fragment of hole h(to) is “item-object” which evaluates to ∅ (h(eats) is unchanged), then we immediately know that stateTransition evaluates to the same result as when hole 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
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., $k_1 = 10$ in Figure 7), and we set the maximum number of combinations of candidate fragments to explore to 10,000 (per level of holes). For the base-choose search strategy, we set the maximum number of candidate fragments to explore for each hole to 1000. The $\#Ast$ column shows whether the fix requires mutations (M) or synthesis (S). Additionally, the authors manually inspect the generated tests and add some new tests to cover different corner cases. The manually written tests account for $\leq 7\%$ of the total tests.

We validate the correctness of a generated patch by both inspecting it 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-choose 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 patch. $✗$ means the patch is plausible (incorrect but passes all tests). $✗$ means ARepair fails to generate a patch that pass all tests. $∞$ means the repair time 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

Figure 9: ARepair Results. Times are in seconds. – denotes not applicable.
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 (≥ 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 (≥ 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 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 \( n \text{.} \text{.} \text{left} \) to represent the domain of \( n \text{.'s} \text{left} \) descendants. The correct domain should be \( n \text{.} \text{left} \text{.} \text{.} \text{left} \text{.} \text{.} \text{right} \) as shown in the human-written patch. The generated patch restricts the domain to be \( n \text{.} \text{left} \text{.} \text{.} \text{right} \) which means all nodes that can be reachable from \( n \) by 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 \( c = c \text{.} \text{ext} \) with \( c \text{ in } c \text{.} \text{.} \text{ext} \). The generated patch uses \( c = c \land c \text{.} \text{.} \text{ext} \) which 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 1link). 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. bemp1.
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 \( (a \Rightarrow b) \land y \); but the correct fix requires \( a \Rightarrow b \lor b \), where \( a, b \) and \( y \) are formulas. stu5 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 technique applies 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 embodies 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


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


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

[34] Xuan-Bach D. Le, David Lo, and Claire Le Goues. 2016. Using Alloy to Support Feature-based DSL Construction for Mining Software Repositories. In ICST.


[37] Yoav Zibin, Michael D. Ernst, and Martin C. Rinard. 2006. Inference and enforcement of data structure consistency specifications. In FSE.

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


[40] Friedrich Steinmann, Marcus Frenkel, and Rui Abreu. 2013. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In ICST.


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

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