Refining Alloy-Based Mutation Operators to Reflect Common Mistakes

Ana Jovanovic
University of Texas at Arlington
Arlington, TX USA
ana.jovanovic@mavs.uta.edu

Mohammad Nurullah Patwary
University of Texas at Arlington
Arlington, TX USA
mxp9161@mavs.uta.edu

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

***Abstract*—Alloy is a declarative modeling language that is well-suited for verifying system designs. A key strength of Alloy is its scenario-finding toolset, the Analyzer, which allows users to explore all valid scenarios that adhere to the model's constraints up to a user-provided scope. Despite the Analyzer, writing correct Alloy models remains a difficult task, partly due to Alloy's expressive operators, which allow for succinct formulations of complex properties but can be difficult to reason over manually. One recent body of work introduces mutation testing for Alloy models, that can also automatically generate a mutant-killing test suite. Unfortunately, a recent empirical study highlights that the existing first order mutant operators are not reflective of common mistakes. Therefore, this paper explores ways in which we can augment the mutation testing process to generate mutants that better reflect actual mistakes that users might make.**

***Index Terms*—Alloy, Mutation Testing, Test Generation**

I. Introduction

Our lives are increasingly dependent on software systems. However, these same systems, even the most safety-critical ones, are notoriously buggy. Therefore, there is a growing need to produce reliable software while keeping the cost low. One solution is to make use of declarative modeling languages to help improve software correctness. Alloy [14] is one such popular modeling language. A key strength of Alloy is the ability to develop models in the Analyzer, an automated analysis engine that invokes off-the-shelf Boolean satisfiability (SAT) solvers to search for scenarios, which are assignments to the sets of the model such that all executed formulas hold. Alloy models and their corresponding scenarios have been used to validate software designs [18], [20], to test and debug code [10], [19], to repair program states [22], [30] and to synthesize security attacks [26], [4], [6].

Unfortunately, the model itself needs to be correct to gain these many benefits. While the Analyzer enables automated analysis of models, the Analyzer only supports ad-hoc techniques for testing the correctness of the model itself, such as enumerating all scenarios and visually inspecting them for issues, which is both time-consuming and error-prone. To address this gap, prior work created AUnit to give users a way to systematically check for the correctness of Alloy models [25]. With the existence of AUnit, several traditional imperative testing practices were ported to Alloy, including mutation testing, fault localization, and repair [27], [28], [29].

μAlloy is the mutation testing framework, which generates mutants, generates a mutant-killing test suite, and performs traditional mutation testing [24], [28], [16]. To generate mutants, μAlloy first defines a series of mutation operators that focus on making manipulations to Alloy constraints at the abstract syntax tree (AST) level. During the mutation generation process, μAlloy takes advantage of Alloy's expressive logic and declarative execution environment to proactively prune equivalent mutants. Moreover, for all non-equivalent mutants, μAlloy generates and stores an AUnit test case that kills the mutant. To perform mutation testing, μAlloy takes the set of mutants generated, an Alloy model, and an AUnit test suite, and as output, reports a mutation score that coveys how many mutants the test suite successfully kills.

However, these mutation operators are based on Alloy's grammar divisions. For instance, when a mutant replaces an operator, the replacement options are all operators in the same division, i.e. for `=`, the possible replacements are all other comparison operators (`!=`, `in`, and `!in`). Traditionally, mutant operators are intended to mimic small mistakes users might realistically make. Unfortunately, a recent user study into mistakes users make when writing Alloy models found that μAlloy's current mutation operators are only able to correct 10.96% of the faulty models. In other words, μAlloy's current mutation operators do not reflect mistakes users actually make.

The main reason for this is that while users often make small logical mistakes, these mistakes manifest as multi-step not single-step mistakes. For instance, when trying to express that two sets have an overlap in elements, a common user mistake is express that one set is a subset of the other ("`a in b`") instead of expressing that there is some intersection between the two sets ("`some a & b`"). However, transforming "`a in b`" into "`some a & b`" requires a second order mutant. Therefore, this paper explores two avenues to generate mutants that better reflect mistakes. First, we use the recent user study to translate common mistakes into new mutation operators. Second, we generate second order mutants at large. In our experimental evaluations, we investigate the effectiveness of both of these approaches to produce an improve set of mutants.

In this paper, we make the following contributions:

New Mutation Operators: We use recently identified common encoding mistakes to define new mutation operators.

Second Order Mutants: We outline a process for generating

(a)

```
1. sig Person {
2.   teaches: set Course,
3.   enrolled: set Course,
4.   projects: set Project
5. }
6. sig Professor,Student in Person {}
7. sig Course {
8.    projects: set Project,
9.    grades: Person -> Grade
10. }
11. sig Project {}
12. sig Grade {}
13.
14. pred inv8 { // A professor cannot teach herself
15.    all p: Person | no p.teaches & p.enrolled
16. }
17. run inv8 for 3
```

(b)

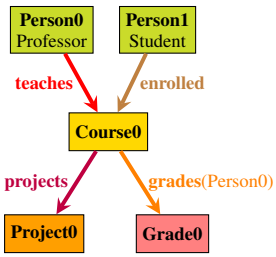

Fig. 1. Alloy model of a course management system and a corresponding satisfying scenario.

second order mutants of Alloy models.

Evaluation: We evaluate the improvements our new mutant operators can achieve in comparison to the increase in overhead. We also investigate the feasibility of second order mutants and highlight why they are beneficial but expensive.

Open Source: We release an implementation of our framework that is built on top of version 6.0.0 of the Analyzer at https://anonymous.4open.science/r/MuAlloyExpansion-FDD5/.

II. Background

In this section, we describe Alloy and μAlloy.

A. Alloy

To highlight how modeling in Alloy works, Figure 1 (a) depicts a model of a course management system. Signature paragraphs and the relations declared within introduce atoms and their relationships (lines 1 - 12). Line 1 introduces `Person` as a named set of atoms and declares three relations (`teaches`, `enrolled`, and `projects`). The relation `teaches` is a binary relation that conveys the idea that each `Person` atom can connect to any number of (`set`) `Course` atoms. Signatures can be declared as extensions or subsets of other signatures. For instance, line 6 introduces the named sets `Professor` and `Student` as subsets of `Person`.

Predicate paragraphs introduce named formulas that can be invoked elsewhere (lines 14 - 16). The `inv8` predicate uses universal quantification (`all`) and set intersection (`&`) to express the idea "no person can be both teaching and enrolled in the same course." Commands indicate which formulas to invoke and place an upper bound on what scope to explore (line 17). The command in Figure 1 (a) instructs the Analyzer to search for a scenario using at most 3 `Person` atoms, 3 `Course` atoms, 3 `Project` atoms and 3 `Grade` atoms such that `inv8` evaluates to true.

Figure 1 (b) displays one such satisfying scenario found by the Analyzer. Scenarios in Alloy are created by making assignments to the sets in the model, where each signature and relation is represented as their own set. In this paper, we refer to the elements within a set as atoms. In Figure 1 (b), `Person` is a set, and `Person0` and `Person1` are the atoms in the set. A user can step through all the scenarios found by the SAT solver, inspecting them for correctness. If no satisfying scenario is found, then the formulas invoked are unsatisfiable for the user-specified scope.

TABLE I
μAlloy Mutant Operators.

Operator	Description
MOR	Multiplicity Operator Replacement
QOR	Quantifier Operator Replacement
UOR	Unary Operator Replacement
BOR	Binary Operator Replacement
LOR	List Operator Replacement
POI	Prime Operator Insertion
UOI	Unary Operator Insertion
VOI	Variable Operator Insertion
BOD	Binary Operator Deletion
UOD	Unary Operator Deletion
PBD	Paragraph Body Deletion
POD	Prime Operator Deletion
LOD	Logical Operand Deletion
VOD	Variable Operator Deletion
BOE	Binary Operand Exchange
IEOE	Imply-Else Operand Exchange

B. μAlloy

μAlloy introduces mutation testing for version 6.0 and earlier of Alloy [24], [28], [16], which focuses on relational, first-order logic, set theory and linear temporal logic.

1) Mutant Generation and Automated Test Generation: μAlloy applies mutation operators to Alloy AST nodes. The currently supported mutation operators can be seen in Table I. Replacement mutant operators will swap logical operators that fall into the same classification in Alloy's grammar, e.g. replacing set intersection "`A & B`" with set union "`A + B`." Exchange mutant operators will swap the order of operands for logical operators with multiple operands, e.g. "**`A`** `&` **`B`**" mutates to "**`B`** `&` **`A`**." Insertion mutant operators will add logical operators, e.g. inserting the reflexive transitive closure operator mutates "`A & B`" to "`*A & B`." Deletion mutant operators will delete logical operators, e.g. deleting the empty set operator in "**`no`** `A & B`" results in the mutant "`A & B`."

For every node in the AST, μAlloy applies all applicable mutation operators one at a time to that location, generating a

series of first-order mutants. Then, μAlloy stores a collection of mutated models that (1) successfully compile and (2) are not equivalent to the original. Unlike mutation testing for imperative languages, μAlloy is able to systematically check at generation if a mutant is equivalent to the original model. In Alloy, `check` commands search for counterexamples, a scenario in which the invoked formulas fail to hold true. Therefore, to determine if a mutant is equivalent, μAlloy executes a command of the form:

```
check {OriginalFormula <=> MutatedFormula}
```

which uses the bi-conditional operator (`<=>`) to assert that the mutated and original formulas should never differ in their truth values. If a counterexample is found, then μAlloy determines the mutant is not equivalent and saves the mutant. In addition, μAlloy will automatically turn the counterexample into an AUnit test case. The end user then labels the converted counterexample as "valid" or "invalid" to provide the oracle for the test case. If no such counterexample can be found, then μAlloy determines the mutant is equivalent and prunes the mutant. In the end, the mutant generation process outputs (1) a set of all non-equivalent mutants and (2) a test suite that is capable of killing all non-equivalent mutants.

2) Mutation Testing: To perform mutation testing, μAlloy takes as input an Alloy model, an AUnit test suite, and a set of mutants. As output, μAlloy reports the mutation score, which displays the ratio of the number of killed mutants to the total number of mutants. μAlloy considers a mutant to be killed if a test case execution passes the mutant model and fails on the original model or vice versa.

III. Mutant Generation Augmentations

In this section, we outline our new mutant operators and a process for generating second order mutants.

A. New Mutant Operators

Our new mutant operators are based on common mistakes made by novice users [17]. Overall, this study identified 16 common mistakes, which we have distilled into 10 new mutation operators, as some common mistakes are already covered by existing mutant operators. In addition, not all mistakes translated to a mutation template, such as "trying to explicitly outlined steps" which meant that users tried to employ a range of temporal operators to outline step by step changes to sets. What this looks like formulaically is very specific to the property being expressed and does not lead to a generic template to apply to any given formula. To illustrate how these mutant operators work, we use the course management model from Section II.

B. First Order Mutants

Signature Extension Replacement (SER): This mutation operator replaces a signature with a parent or child signature. Signatures can be declared as subset (`in`) or extensions of (`extends`) other signatures. When this happens, the child signature is a subtype of the parent signature. As a result, parent and children signatures can be exchanged without causing a type error.

```
from: all s: Student | s.projects in s.enrolled.projects
  to: all s: Person | s.projects in s.enrolled.projects
```

In this example, the domain become more broad, while the mutation operator unfolding in the opposite direction would make the domain more narrow.

Field Replacement (FER): This mutation operator replaces a field with other fields form the same signature.

```
from: all p: Person | p in teaches
  to: all p: Person | p in enrolled
```

Depending on the domain being modeled, field declarations can represent similar subsets of atoms that all have the same internal type, which can result in users accidentally interchanging the fields.

Nested Quantification Disjoint Insertion (NQDI): This mutation operator inserts the disjoint ("`disj`") keyword for nested quantification. If nested operators reason over overlapping sets for their domain, then the disjoint keywords enforces that the quantified variables can never be assigned the same atom at the same time.

```
from: all p1, p2 : Person | p1.teaches != p2.teaches
  to: all disj p1, p2: Person | p1.teaches != p2.teaches
```

Nested Quantification Exchange (NQE): When the nested quantification operators are different, the order in which the quantification is declared impacts the valid behavior, even if the nested quantified occurs as back to back declarations. Therefore, this mutation operator swaps the order the nested quantification is declared in.

```
from: one c: Course | all p: Project | c->p in projects
  to: all p: Project | one c: Course | c->p in projects
```

In this case, the first formula checks that for exactly one course atom, all project atoms are projects associated with this course. In contrast, the second formula checks that for all project atoms, there is exactly one course that contains this project.

C. Higher Order Mutants

Quantifier Domain Insertion (QDI): This operator changes the quantified domain if the domain originally consists of a signature that has one or more children signatures, irregardless of if they are subset or extension signatures. This operator manipulates the domain by insertion the binary set operators (difference (`-`)), insertion (`&`)), union (`+`)) along with the children signatures.

```
from: all t: Person | some t.Teaches
  to: all t: Person - Student | some t.Teaches
  to: all t: Person - Teacher | some t.Teaches
```

This operator is particularly helpful when a parent signature has multiple children.

When looking into common mistakes that users make when writing formulas, we found that users often did not know when

to use quantification operators versus when to use relational multiplicity operators. At a high level, this means that users frequently did not make the right choice when trying to check whether a property holds for all elements of a set or whether a property holds for a set at large. To address this, we introduce two higher order mutant operators.

Quantifier Operator Deletion (QOD): This mutation operator replaces a quantified formula with a generalized relational formula.

```
from: all t:Teacher| some t.Teaches
  to: some Teacher.Teaches
```

Quantifier Operator Insertion (QOI): This mutation operator replaces a generalized relational formula with a quantified formula.

```
from: some Teacher.Teaches
  to: all t:Teacher| some t.Teaches
```

Set Subset Replacement (SSR): This mutation operator interchanges the subset with a formula that checks for some set intersection. This operator also interchanges the subset exclusion operator with a formula that checks for no set intersection. At a high level, this mutation operator address a subtle different when trying to check if there is some or no overlap between two sets. The biggest way that these two formulations differ is when reasoning over empty sets. For empty sets, "`a in b`" will be true but "`some a & b`" will be false. This can produce incorrect behavior where the model may fail to generate scenarios where these sets are empty.

```
s: all p : Person | p.teaches !in p.enrolled
c: all c: Component | no p.teaches & p.enrolled
```

Set Disjoint Replace (SDR): This mutant operator ensures that when checking if an atom is included in only one of two sets, that the atom is not actually in both sets, ensuring the desired disjoint-ness.

```
from: all p : Person | p in Professor or p in Student
  to: all p : Person | p in Professor => p !in Student
```

This mutation operator is similar to **SSR** and is rooted in the same common mistakes. but focuses on how this mistake manifests when the user is trying to check that an atom is only in one of two sets. For instance, in English saying "a person is either a professor or a student" implies a person is not both. However, simply using the disjunction operator (`or`) allows for a person to be both, despite the keyword matching the English sentence.

Unary Temporal Exchange (UTE): This mutation operator moves a leading temporal operator inside the quantified formula. The course management example does not contain mutable signatures or relations. Therefore, the below example is from the trainstation_ltl model. The property is checking that for a train track, every signal will eventually turn green.

```
from: eventually all s: Signal |  s in Green
  to: all s: Signal | eventually s in Green
```

Algorithm 1: Second Order Mutant Generation

Input: Alloy model *module*, Mutation Operators *muOps* organized by type (D,E,I,R)
Output: Second order mutants of *module*.

```
1  firstOrderMutants ← map type (D,E,I,R) to mutants
2  while module.hasMoreASTNodes() do
3      node = module.nextASTNode()
4      for muOp ∈ muOps do
5          if canApplyMutationOperator(node, muOp then
6              firstOrder = applyOperator(node, muOp)
7              if isValid(firstOrder) then
8                  firstOrderMutants.put(muOp.type,firstOrder)
9  secondOrderMutants ← map type (DD,DE,etc) to mutants
10 for type1 ∈ firstOrderMutants.keySet() do
11     for firstOrderMutant ∈ firstOrderMutants.get(type1) do
12         while firstOrderMutant.hasMoreASTNodes() do
13             node = firstOrderMutant.nextASTNode()
14             for muOp ∈ muOps do
15                 if canApplyMutationOperator(node, muOp
                    then
16                     secondOrder = applyOperator(node,
                        muOp)
17                     if isValid(secondOrder) then
18                         secondOrderMutants.put(type1 +
                            muOp.type,secondOrder)
19 return secondOrderMutants
```

The novice study highlights that users had a high preference to place all the temporal operators at the front of the formula. However, which subformulas the temporal operator applies to can have a subtle but meaningful impact on if a boundary scenario is valid or not.

D. Second Order Mutants

The recent empirical study over novice mistakes highlights that small conceptual mistakes in encoding logic more often translate to multi-step edits to fix. Therefore, we also want to explore the viability of second order mutants. At a high level, to generate a second order mutant, we first use μAlloy to produce all first order mutants for a model m. Then, for each first-order mutant f, we run μAlloy on the mutant, producing first order mutants of f, which are then second-order mutants of m.

Algorithm 1 shows our algorithm for generating second order mutants in detail. Lines 1 - 8 follow μAlloy's traditional mutant generation approach. The only difference is that we now store the first order mutants by their type ('`D`' for deletion mutants, '`E`' for exchange mutants, '`I`' for insertion mutants, and '`R`' for replacement mutants). This mapping, stored in `firstOrderMutants`, is then used as a base to generate second order mutant.

Lines 9 - 19 generate and return all valid, non-equivalent second order mutants. We first loop over all first order mutants by type (line 10 - 11). We then apply the standard mutant generation process where we apply all applicable mutant operators for all AST nodes (lines 12 - 16). Then, for each

mutant produced by a mutation operator, we check is that mutant (`secondOrder`) is valid and non-equivalent to the original model (`module`). If so, we then save the second order mutant (lines 17-18). At the end, we return all second order mutants organized by type (line 19). The type '`DD`' would mean the second order mutant was the result of two deletion mutation operators being performed.

IV. Evaluation

We address the following research questions:

- **RQ1:** What is the overhead of our expanded mutation operators compared to the original operators?
- **RQ2:** How effective are our expanded mutation operators at detecting faults compared to the original operators?
- **RQ3:** How effective are our expanded mutation operators at correcting faults compared to the original operators?
- **RQ4:** What is the tradeoff in potential versus overhead of second order mutants?

A. Set Up

In our study, we use the publicly released Alloy4Fun dataset, which contains real-world models obtained from master's students submissions from the University of Minho (UM) and the University of Porto (UP) in the academic period from Fall 2019 to Spring 2023 [5]. For these models, users were tasked with completion predicates in a given Alloy model to match a provided English description. In total, there are 97,755 submissions that span 17 different Alloy models and 183 predicates to be filled in (exercises). We filtered out correct submissions, non-compilable submissions and syntactic duplicates, leaving 25,180 unique faulty submissions.

Table II gives an overview of the complexity of the models used in our study in terms of the universe of discourse each model creates. Column **Model** is the Alloy4Fun model under consideration. Models with an underscore in their name represent models that have multiple versions in the dataset. Between versions, the number of exercises, instructional text, and/or the type of logic can change. Column **#Sig** is the total number of signatures in the model, **#Abs** is the number of abstract signatures, **#Ext** is the number of signatures that extend another signature, **#Rel** is the number of relations, **Arity** is the average arity of all relations in the model (2 indicates a binary relation), **#Exe** is the number of exercises and **#AST** is the average number of abstract syntax tree (AST) nodes in the oracle solutions for all exercises of that model. Column **#Faults** is the number of faulty submissions for that model. A faulty submission consists of the base model (signature and fact paragraphs) and the executed faulty predicate. Given this composition of a faulty submission, all results presented represent the performance of generating mutants, generating tests and performing mutation testing for a single predicate within a model.

TABLE II
Complexity of Base Models

Model	#Sig	#Abs	#Ext	#Rel	Arity	#Exe	#AST	#Faults
classroom_fol	5	0	2	3	2.33	15	10.00	1499
classroom_rl	5	0	2	3	2.33	15	10.13	1286
courses_v1	6	0	2	5	2.2	15	16.87	3948
courses_v2	6	0	2	5	2.2	15	16.87	2210
cv_v1	5	1	2	4	2	4	19.75	404
cv_v2	5	1	2	4	2	4	21.75	168
graphs	1	0	0	1	2	8	7.63	816
lts	3	0	1	1	3	6	19.71	761
productionLine_v1	5	0	2	3	2	4	14.25	191
productionLine_v2	10	1	7	4	2	10	14.90	1495
productionLine_v3	10	1	7	4	2	10	14.90	1091
socialMedia	5	0	2	5	2	8	15.75	5493
trainstation_fol	7	0	5	2	2	10	13.40	2569
trainstation_ltl	6	0	4	3	2	17	23.44	608
trash_fol	3	0	2	1	2	10	4.80	436
trash_ltl	3	0	2	1	2	20	8.55	1630
trash_rl	3	0	2	1	2	10	4.80	575
AVG/TOTAL	**5.18**	**0.24**	**2.71**	**2.94**	**2.12**	**10.65**	**13.97**	**25180**

B. RQ1: Overhead Comparison

Tables III and IV present an overview of the performance of mutant generation and mutation testing. Table III highlights the performance for the original mutant operators, our baseline. Table IV highlights the performance with our new mutant operators included. Column **Model** is the Alloy4Fun exercise under consideration. As Table II highlights, there are multiple submissions per exercise. Therefore, we present the average, min and max for all of our metrics. The next four sections present metrics for the mutant generation process. The columns under **#EQ** is the number of equivalent mutants and the columns under **#NEQ** is the number of non-equivalent mutants. During generation, we produce a test suite that kills all non-equivalent mutants. The columns under **#Test** displays the number of tests produced. Lastly, the columns under $\mathbf{T}_{gen}$ is the overall runtime to generate mutants, which includes checking for equivalence and capturing test cases to kill all non-equivalent mutants. For mutation testing, we use this test suite. As a result, the mutation score is, by design, always 100%; therefore, we do not report the mutation testing score. The final division, $\mathbf{T}_{mt}$, is the runtime to perform mutation testing. All runtimes are shown in seconds.

μAlloy has previously only been evaluated over a small benchmark of 13 models [28] and the two temporal Alloy4Fun submissions sets (**trash_ltl** and **trainstation_ltl**). Therefore, we first highlight some of the common trends that highlight characteristics of μAlloy at large. Often times, the number of tests needed to kill the non-equivalent mutants is notable less than the number of non-equivalent mutants. As a result, non-equivalent mutants do not produce a linear increase in work for the user to provide an oracle for the generated test. In addition, the average runtime to generate mutants and perform mutation testing is trivial, with generating mutants taking about 3.5 seconds and mutation testing taking 3 to 5 seconds.

However, there are outliers. For four models (**courses_v2**, **socialMedia**, **trainstation_ltl** and **trash_ltl**) the maximum runtime encountered for mutant generation is 22 to 323 minutes. This spike in runtime occurs when the faulty submission under consideration takes the SAT solver a long time to solve. In generation, this faulty submission is analyzed repeatedly to determine if a mutant is equivalent. For mutation testing, these problematic faulty submissions also bloat the runtime, although notably with less impact as (1) the faulty predicate is being invoked less often and (2) test cases narrow the SAT search by outlining explicit set restrictions for all the signatures and relations in the model.

TABLE III
BASELINE MUTATION GENERATION AND TESTING PERFORMANCE

Model	# EQ			# NEQ			# Tests			$\mathbf{T}_{gen}$			$\mathbf{T}_{mt}$		
	avg	min	max	avg	min	max	avg	min	max	avg	min	max	avg	min	max
classroom_fol	5.83	0	93	22.72	4	108	12.42	1	57	0.80	0.06	3.80	1.24	0.06	16.72
classroom_rl	4.20	0	116	16.18	3	111	8.38	1	39	0.48	0.06	4.21	0.82	0.04	17.73
courses_v1	6.32	0	84	20.66	4	72	9.36	1	43	2.03	0.20	8.55	1.69	0.15	12.25
courses_v2	7.54	0	109	20.74	1	126	9.20	1	57	2.52	0.19	1766.66	2.26	0.05	2108.41
cv_v1	8.68	0	47	19.26	3	52	6.57	1	20	0.72	0.06	1.93	0.76	0.03	3.49
cv_v2	8.04	0	37	21.31	6	55	7.14	2	26	0.83	0.21	2.42	0.83	0.10	3.65
graphs	9.44	0	64	18.11	2	67	5.90	1	23	0.52	0.05	2.14	0.17	0.01	1.75
lts	4.18	0	36	12.92	3	43	6.01	1	24	0.48	0.06	8.86	0.28	0.03	3.15
productionLine_v1	3.13	0	26	12.83	2	44	5.43	1	16	1.02	0.14	2.99	0.47	0.05	2.38
productionLine_v2	13.41	0	117	24.14	2	86	13.08	1	63	1.19	0.24	5.95	3.05	0.10	27.65
productionLine_v3	13.40	0	101	25.22	2	130	13.02	1	48	1.15	0.17	6.33	3.02	0.07	23.61
socialMedia	12.09	0	104	30.13	3	141	24.11	1	123	4.45	0.50	17254.14	2.41	0.04	787.25
trainstation_fol	7.33	0	110	24.40	1	102	12.63	1	51	0.69	0.54	3.86	1.72	0.05	30.10
trainstation_ltl	27.17	2	171	40.45	3	181	24.61	1	132	34.73	0.65	1603.61	50.86	0.12	3299.96
trash_fol	5.49	0	40	17.52	2	37	7.98	1	22	0.40	0.05	1.21	0.43	0.02	1.91
trash_ltl	16.97	2	99	39.58	5	93	23.31	1	67	20.95	0.62	1998.76	2.75	0.09	17.82
trash_rl	4.19	0	53	13.61	2	47	6.37	1	17	0.39	0.06	4.41	0.32	0.02	4.67
TOTAL	9.65	0	171	24.53	1	181	14.31	1	132	3.54	0.05	17254.14	2.56	0.01	3299.96

TABLE IV
NEW OPERATORS MUTATION GENERATION AND TESTING PERFORMANCE

Model	# EQ			# NEQ			# Tests			$\mathbf{T}_{gen}$			$\mathbf{T}_{mt}$		
	avg	min	max	avg	min	max	avg	min	max	avg	min	max	avg	min	max
classroom_fol	12.30	0	117	45.40	5	149	19.84	2	74	1.26	0.09	5.01	2.80	0.09	24.03
classroom_rl	8.21	0	170	31.94	4	166	13.30	2	54	0.66	0.07	4.84	1.81	0.06	33.92
courses_v1	13.09	0	118	38.16	4	103	15.15	1	56	2.27	0.16	8.16	3.57	0.15	24.44
courses_v2	14.56	0	149	37.95	1	162	14.82	1	69	2.83	0.14	1885.09	4.44	0.05	3478.52
cv_v1	13.89	0	64	30.60	4	70	9.12	1	25	1.11	0.17	3.48	1.23	0.03	4.66
cv_v2	13.86	0	50	31.76	7	71	9.59	2	30	0.88	0.31	3.10	1.24	0.11	4.53
graphs	10.41	0	68	20.15	2	69	6.24	1	23	0.37	0.03	1.73	0.14	0.01	1.34
lts	7.29	0	44	22.27	4	74	9.09	2	29	0.47	0.05	8.38	0.53	0.04	6.89
productionLine_v1	7.24	0	46	21.09	4	61	7.41	2	21	1.14	0.18	3.36	0.77	0.09	3.31
productionLine_v2	27.32	0	241	56.39	2	185	22.39	2	106	1.58	0.13	7.12	7.81	0.11	55.56
productionLine_v3	26.68	0	248	56.45	2	186	21.53	1	96	1.53	0.20	6.13	7.38	0.07	43.12
socialMedia	19.50	0	143	52.59	3	243	34.53	2	154	5.11	0.08	19415.63	4.72	0.06	1443.40
trainstation_fol	14.72	0	164	52.92	1	224	20.29	1	85	1.02	0.07	5.85	4.14	0.05	49.24
trainstation_ltl	32.96	2	188	55.46	6	212	28.41	2	140	29.84	0.52	1346.15	73.73	0.32	4724.06
trash_fol	10.48	0	58	27.98	5	64	10.56	1	33	0.47	0.07	1.36	0.72	0.04	3.10
trash_ltl	21.36	2	120	50.19	8	111	26.79	2	77	18.58	0.58	1743.76	3.50	0.11	19.96
trash_rl	7.30	0	67	22.95	2	53	8.97	1	26	0.47	0.08	4.82	0.68	0.02	31.08
TOTAL	16.58	0	248	44.16	1	243	21	1	154	3.6	0.03	19415.63	4.93	0.01	4724.06

The other common thread between the base operators and our expanded operators is that first order mutants do frequently create equivalent mutants. Equivalent mutants make up an average of 39.34% of all mutants generated for the base operators and an average of 27.29% for the expanded operators, which does include some higher order mutants. In other words, making a single change to a formula has a high likelihood of not actually changing the behavior of the formula. Overall, this reinforces the idea that when a user makes a faulty formula, the corrections needed are likely to require multiple edits.

> In terms of overhead, more non-equivalent mutants does not lead to a one-to-one increase in tests and the runtimes associated with mutation testing are often minor. However, first order mutants are also likely to simply be equivalent to the original formula.

While there are high level trends about μAlloy that our experiments help highlight, there are still differences in behavior between the original mutant operators and the expanded mutant operators due to the increase in the number of operators that can be considered at a given AST node. Therefore, to help compare these performances, Table V presents the difference in average performance when subtracting μAlloy's average performance with the new mutant operators from μAlloy's baseline average performance. The columns **Model**, **#NEQ**, **#EQ**, **#Test**, $\mathbf{T}_{gen}$ and $\mathbf{T}_{mt}$ all have the same meaning as Tables III and IV.

Since we have added mutation operators, we expect that the number of mutants generated, the number of tests and the runtimes to all increase. These trends are supported in Table V. However, there are a few models in which the generation time is faster for the new mutant operators. There are a few reasons why this occurs. First, some of the new mutant operators rely on certain properties to be present in the model to apply, such as extension signatures, nested quantification or temporal logic. Therefore, for some models, there will be a subset of the new mutant operators that will never apply. Second, when solving back to back formulas that are similar, Alloy's SAT solver is primed from the first invocation and can

TABLE V
COMPARISON BETWEEN BASELINE AND NEW OPERATORS

Model	# EQ	# NEQ	Diff N-O # Tests	T_{gen}	T_{mt}
classroom_fol	6.47	22.68	7.42	0.46	1.56
classroom_rl	4.01	15.76	4.92	0.18	0.99
courses_v1	6.77	17.50	5.79	0.24	1.88
courses_v2	7.02	17.21	5.62	0.31	2.18
cv_v1	5.21	11.34	2.55	0.39	0.47
cv_v2	5.82	10.45	2.45	0.05	0.41
graphs	0.97	2.04	0.34	-0.15	-0.03
lts	3.11	9.35	3.08	-0.01	0.25
productionLine_v1	4.11	8.26	1.98	0.12	0.30
productionLine_v2	13.91	32.25	9.31	0.39	4.76
productionLine_v3	13.28	31.23	8.51	0.38	4.36
socialMedia	7.41	22.46	10.42	0.66	2.31
trainstation_fol	7.39	28.52	7.66	0.33	2.42
trainstation_ltl	5.79	15.01	3.80	-4.89	22.87
trash_fol	4.99	10.46	2.58	0.07	0.29
trash_ltl	4.39	10.61	3.48	-2.37	0.75
trash_rl	3.11	9.34	2.60	0.08	0.36
TOTAL	6.93	19.63	6.69	0.06	2.37

solve the second one faster. Since mutant generation involves solving many overlapping SAT problems, the increase in the number of mutants being evaluated does not always have a net negative effect. Third, SAT solvers have variability in the paths they explore first when trying to solve a CNF formula. This variability can lead to small differences in runtimes even when evaluating the same formula back to back.

For instance, the generation time for **trainstation_ltl** submissions is on average 4.89 seconds faster with the new mutant operators. This is likely because there are less submissions for this model, therefore, the SAT solver got more primed due to the increase in mutants. However, this does not translate to the mutation testing runtime, where **trainstation_ltl**'s average runtime is 23 seconds longer with the new mutant operators considered. This is because test cases do not produce the overlapping subproblems that mutants do, as test cases can vary drastically from each other. All told, the runtimes are largely similar, with often a nominal different in mutant generation time and a small difference in mutation testing time.

Moreover, while about 20 more non-equivalent mutants are generated, this only amounts to an average of 6 additional tests. The number of tests reflects the manual labor a user has to put into mutation testing, as the user does need to provide the test oracle. However, labeling an additional six tests is not likely to produce a noticeable increase in burden on the user on top of the average of 14 tests they would already be labeling.

> While the new mutant operators do increase both the number of mutants generated and the number of tests, the overall impact on total overhead is small.

C. RQ2: Detection Comparison

While our mutant killing test suite is capable of detecting all non-equivalent mutants, that does not mean that our test suite will detect all faults in a model. We consider a mutant to detect a fault when at least one test case behaves differently on the mutated model compared to the oracle model. This differs from correction, in which the mutated model is logically equivalent to the oracle model. Table VI displays the results of this experiment. Column **Model** is the Alloy4Fun exercise and column **#T** is the total number of unique faulty submissions for the exercise. The next two columns depict the detection ability of the original mutant operators. Column **#D** is the number of faulty submissions that were detected and **%D** is the percentage. The next two columns present the same information when our new mutant operators are considered. Lastly, column **% Change** depicts the increase in detection our new operators enables, which is calculated by subtracting the new mutant operator **%D** from the original.

TABLE VI
ABILITY OF MUTANTS TO DETECT FAULTS

Model	#T	Orignal #D	Orignal %	New Operators #D	New Operators %	% Change
classroom_fol	1499	1398	93.26	1428	95.26	2.00
classroom_rl	1286	1149	89.35	1211	94.17	4.82
courses_v1	3948	3699	93.69	3775	95.62	1.93
courses_v2	2210	2067	93.53	2109	95.43	1.90
cv_v1	404	173	42.82	371	91.83	49.01
cv_v2	168	56	33.33	152	90.48	57.14
graphs	816	743	91.05	754	92.40	1.35
lts	761	702	92.25	722	94.88	2.63
productionLine_v1	191	160	83.77	167	87.43	3.66
productionLine_v2	1495	1371	91.71	1408	94.18	2.47
productionLine_v3	1091	1000	91.66	1026	94.04	2.38
socialMedia	5493	5407	98.43	5456	99.33	0.89
trainstation_fol	2569	2493	97.04	2523	98.21	1.17
trainstation_ltl	608	253	41.61	538	88.49	46.88
trash_fol	436	386	88.53	395	90.60	2.06
trash_ltl	1630	1546	94.85	1555	95.40	0.55
trash_rl	575	508	88.35	529	92.00	3.65
TOTAL	**25180**	**23111**	**91.78**	**24119**	**95.79**	**4.00**

The original mutant operators are fairly effective at detecting faults: on average, μAlloy's original operators detect 91.78% of all faulty submissions. Notably, the performance is lagging for three models: **cv_v1**, **cv_v2** and **trainstation_ltl**. These three models do have the longest oracle solutions, which implies that their solutions are more complex. In addition, these models also have less submissions made for them overall. The high level of difficulty may turn off inexperienced users from attempting these submissions. This could increase the degree of which the faulty submission is close to the oracle solution, making detection harder for these models as the test cases would be more likely to behave the same for both models. Importantly, our new mutant operators close this gap for these models w.r.t the other models in the dataset. The average detect rate with the new operators is 95.79%, with these three models have a minimum detection rate of 88.49%. In addition, for **socialMedia**, the new mutant operators raise the detection rate to over 99%.

> The new mutant operators address the gaps in the fault detection capability of the previous set of operators, given the drastic increase in fault detection ability the new mutant operators achieve with respect to the models with less than a 50% detection previously.

D. RQ3: Correction Comparison

Our motivation for creating new mutant operators is that the original mutant operators did not correlate to common mistakes that users make. Therefore, we want to evaluate

TABLE VII
ABILITY OF MUTANTS TO CORRECT FAULTS

Model	#T	Orignal		New Operators		% Change
		#C	%	#C	%	
classroom_fol	1499	114	7.61	207	13.81	6.20
classroom_rl	1286	114	8.86	221	17.19	8.32
courses_v1	3948	150	3.80	659	16.69	12.89
courses_v2	2210	91	4.12	420	19.00	14.89
cv_v1	404	13	3.22	20	4.95	1.73
cv_v2	168	8	4.76	14	8.33	3.57
graphs	816	184	22.55	203	24.88	2.33
lts	761	40	5.26	60	7.88	2.63
productionLine_v1	191	22	11.52	43	22.51	10.99
productionLine_v2	1495	220	14.72	284	19.00	4.28
productionLine_v3	1091	122	11.18	166	15.22	4.03
socialMedia	5493	632	11.51	822	14.96	3.46
trainstation_fol	2569	324	12.61	443	17.24	4.63
trainstation_ltl	608	67	11.02	90	14.80	3.78
trash_fol	436	145	33.26	188	43.12	9.86
trash_ltl	1630	359	22.02	407	24.97	2.94
trash_rl	575	155	26.96	215	37.39	10.43
TOTAL	**25180**	**2760**	**10.96**	**4462**	**17.72**	**6.76**

the effectiveness of our new mutant operators by determining what impact these operators have on our ability to correct faulty models. To evaluate this, for each faulty submission, we checked whether any mutant generated was equivalent to the oracle solution. If so, then we flagged the faulty submission as correctable. Table VII displays the results of this experiment. Column **Model** is the Alloy4Fun exercise and column **#T** is the total number of unique faulty submissions for the exercise. The next two columns depict the correction ability of the original mutant operators. Column **#C** is the number of faulty submissions that were corrected and **%C** is the percentage. The next two columns present the same information when our new mutant operators are considered. Lastly, column **% Change** depicts the increase in detection our new operators enables, which is calculated by subtracting the new mutant operator **%C** from the original.

The original mutant operators can only correct 10.96% of all faulty submissions. There are some models, such as **graphs** and the three trash models (**trash_fol**, **trash_ltl** and **trash_rl**) where the correction rate is over 20%. However, there are also several models in which the correction rate is at or below 5% (**courses_v1**, **courses_v2**, **cv_v1**, **cv_v2**, **lts**). Interestingly, these results highlights that the original mutant operators are capable of generating diverse test suites that can reveal faults, despite the mutants themselves not accurately representing common modeling mistakes. To further highlight that fault detection and correct are not tightly coupled for Alloy models, while the fault detection for **trainstation_ltl** is low, the correction rate for **trainstation_ltl** is higher than several models that had a better fault detection ability.

In comparison, our new mutant operators raise the correction rate to 17.72% on average. With the new mutant operators, only one model, **cv_v1** has a correction rate of less than 5%. At the same time, several models now have a correction rate of over 35%, with **trash_fol** and **trash_rl** having a correction rate of 43.12% and 37.39% respectively. While all models experienced an increase in correction rate, the highest increase is on models where the parent/child signature relationships is present (**classroom_fol**, **classroom_rl**, **courses_v1**, **courses_v2**, **trash_fol**, **productionLine_v1**, **trash_rl**). This is not surprising, as multiple new mutation operators (**SER** and **QDI**) address the existence of extension and subset signatures, while none of the original mutation operators reason over this attribute at all.

TABLE VIII
MUTANT OPERATORS THAT CORRECT MISTAKES

Op	# Fix	Percent
UOI	1270	16.20
UOR	354	4.52
UOD	158	2.02
BOR	824	10.51
BOD	286	3.65
BOE	172	2.19
LOD	218	2.78
LOR	53	0.68
QOR	685	8.74
POI	55	0.70
PBD	0	0.00
SER	1410	17.99
FER	102	1.30
NQE	58	0.74
NQDI	74	0.94
QOD	6	0.08
QOI	0	0.00
SSR	366	4.67
QDI	1715	21.88
SDR	2	0.03
UTE	31	0.40

To get a better understanding of how representative each mutation operator is of real mistakes, Table VIII displays the rate of which a given mutation operator is able to correct a faulty model. Column **Op** is the mutation operator under consideration. Column **#Fix** is the number of mutants of that operator that fix a faulty submission and **Percent** is the percentage that this number is out of the total number of correcting mutants. Of the original operators, **UOI**, **BOR** and **QOR** mutants predominately fixed faulty submissions. Two new mutant operators, **SER** and **QDI**, provide more fixes than any of the original operators. As Table II highlights, a substantial number of the models from Alloy4Fun have extension signatures. This was clearly a gap in the previous mutant operators, and closing this gap is one of the main reasons that the new operators are capable of correcting more mistakes.

> The new mutant operators improve the correction rate, most notably by covering aspects of the language not previously handled.

The nested quantification operators can correct some faults, but not a high amount. However, this is expected. A recent static profile of publicly available Alloy models on Github reveals that nested quantification is rarely used [11]. Furthermore, the **NQDI** operator will only produce a non-equivalent mutant if the nested quantified formulas reason over the same domain and the user forgot to account for the overlap in elements. That said, the **NQE** operator does highlight that users do make mistakes when trying to determine the right order in which to nest quantified formulas.

Three of the new mutant operators are not effective at correcting mistakes: **QOI**, **QOD** and **SDR**. While **SDR** is rooted in a common mistake pattern for how users check set membership, the operator is a more niche version than the

TABLE IX
SECOND ORDER MUTANT CORRECTIONS AND SIZE OVERVIEW

SO	graphs				cv_v1				trash_fol			
	# Fix	% Fix	# Mut	# Tests	# Fix	% Fix	# Mut	# Tests	# Fix	% Fix	# Mut	# Tests
DD	46	5.64	16.17	2.5	3	0.74	45.17	3.41	11	2.52	6.94	1.68
DE	18	2.21	4.42	1.87	0	0.00	11.16	2.22	7	1.61	3.4	2.30
DI	132	16.18	31.37	3.86	0	0.00	33.06	2.36	45	10.32	9.83	3.18
DR	170	20.83	49.56	8.53	299	74.01	177.61	20.11	157	36.01	33.89	8.72
ED	29	3.55	9.47	2.91	0	0.00	21.13	3.72	32	7.34	8.69	3.38
EE	4	0.49	2.07	1.32	0	0.00	4.27	1.43	0	0.00	4.03	1.72
EI	20	2.45	24.3	3.04	0	0.00	19.39	1.74	9	2.06	19.07	4.59
ER	98	12.01	24.31	8.15	49	12.13	65.04	17.56	57	13.07	36.29	11.35
ID	140	17.16	64.06	4.99	0	0.00	77.88	4.53	51	11.70	31.92	4.37
IE	20	2.45	17.37	2.43	0	0.00	19.27	1.60	9	2.06	20.92	4.91
II	112	13.73	106.44	3.92	0	0.00	6.46	1.11	56	12.84	54.63	4.73
IR	240	29.41	158.82	10.37	71	17.57	235.08	15.98	147	33.72	134.42	16.07
RD	163	19.98	66.26	6.07	277	68.56	185.68	11.44	149	34.17	52.89	6.38
RE	90	11.03	15.77	3.98	42	10.40	42.06	5.89	56	12.84	27.84	6.94
RI	222	27.21	130.48	7.41	38	9.41	90.55	5.11	136	31.19	90.1	10.60
RR	257	31.50	160.51	11.08	187	46.29	593.44	70.68	169	38.76	209.54	16.54
Combined	378	46.32	881.38	82.43	308	76.24	1627.25	168.89	280	64.22	744.4	107.46

more successful **SSR** operator. However, **QOI** and **QOD** are derived from users selecting the wrong level of quantification, which the novice study reveals is the most frequent mistake. These mutant operators ended up not being as effective at correcting faults because when the wrong level of quantified is used, the subformula is often very similar but slightly different. Manually inspecting the common mistakes that inspired these mutation operators revealed that the current **QOD** and **QOI** combined with a **BOR** or **UOR** mutant application in the subformula would have corrected many more faults.

> Mutant operators changing the level of quantification additionally highlight that real mistakes often involve more than one location.

E. RQ4: Effectiveness vs. Expense of Second Order Mutants

Although our new mutant operators close the gap in detection, the correction rate of the mutant operators is still low at an average of 17.72%. As a result, there is still a disconnect between the common mistakes users make and our mutation operators. To explore if second order mutants better reflect common mistakes, Table IX displays the fault correction capabilities of second order mutants for the **graphs**, **cv_v1** and **trash_fol** models. The **graphs** model is one of the simplest models in the Alloy4Fun dataset, and can highlight potential scalability issues even for simple models. The **cv_v1** models has the lowest correction rates, while the **trash_fol** has the highest correction rate for first order mutants. Column **SO** is the type of second order mutant, where D is a deletion action, E is an exchange action, I is an insertion action and R is a replacement action. For each model, column **# Fix** is the number of faulty submissions fixed by the type of second order mutant and column **% Fix** is what percentage this is out of the total number of faulty submissions for the model. Column **# Mut** is the number of non-equivalent mutants produced and **# Tests** is the number of test cases needed to kill all non-equivalent mutants. The **Combined** row highlights the performance of all second order mutants considered together.

As Table IX highlights, second order mutants are inherently more representative of the mistakes users makes as across the model. For our subset of three models, 46.32% to 76.24% of the faulty submissions can be directly fixed by a second order mutant. However, producing all second order mutants produces a high overhead. For the simpler **graphs** and **trash_fol** models, which have smaller predicates, there are on average 881.31 and 744.4 second order mutants generated per predicate. Even worse, for the more complex **cv_v1** model, there is on average over 1,600 second order mutants created per predicate. Interestingly, this average number of tests generated to kill these mutants is magnitudes smaller: 82 for **graphs**, 169 for the **cv_v1** model and 107 for **trash_fol**. One reason for this is that there may be high redundancy in second order mutants. We may be able to notably prune the number of mutants generated if we also check for equivalence with any previously existing mutant, not just equivalence with the original formula being mutated. However, filtering out mutants that are equivalent to other mutants is not guaranteed to reduce the number of tests, as those mutants may have already produced redundant tests. Either way, the number of tests created is too large to reasonably expect a user to manually label.

In terms of effectiveness, not all categories of second order mutants are equal. Across all models, any second order mutant combination that involves a replacement action is substantially more representative of mistakes than any of the other combinations. In addition, the remaining insertions second order combinations (**DI**, **ID**, and **II**) are fairly effective for the graph and trash models.

> Second order mutants are promising, as they are much more representative of mistakes users actually make. however, a brute force approach to second order mutants creates an infeasible number of tests to label per predicate for the user.

While second order mutants capture real mistakes users make, second order mutants have a high overhead to generate. As Tables III and IV indicate, there are some first order mutants that take a prohibtive amount of time to generate mutants for. This issue is only compounded for second order mutants, where these problematic first order mutants took days to generate second order mutants for. To illustrate consider the inv8 faulty submission for **trainstation_ltl** shown below which took about 27 minutes to generate first order mutants for:

```
1. pred inv8[] {
2.   always (all t: Train {
3.     ((some t.pos.signal && some t.pos.prox) &&
4.     (t.pos.signal in (Signal - Green))) =>
5.     eventually once (some t.pos) since
6.     (t.pos.signal in Green)
7.   })
8. }
```

In addition to being verbose, this formula contains quantification and nested temporal operators that include a combination of future and past operators. Analyzing this formula, and the resulting second order mutants that can insert more temporal operators, requires extensive computational resources. Specifically, to generate all second order mutants for this submission took about 7 days. Even for less intensive models, the runtime for second order mutants scales up quickly. For example, the small, much less complex **graphs** model has an average runtime of 11.8 seconds to generate second order mutants, while first order mutants only take 0.5 seconds.

> Second order mutants are not scalable, especially if the underlying formula takes a non-trivial amount of time to analyze by itself. All told, this makes second order mutants at large not feasible.

V. Future Work: Potential Avenues for Select Subset of Second Order Mutants

Even though the generation of second order mutants comes with a cost, second order mutants are effective. Therefore, in future work, we plan to explore methods for efficiently creating subsets of valuable second order mutants. One approach we plan to take is to focus on the second order mutant combinations that are the most representative of mistakes, while also pruning mutants that are equivalent to other mutants. As another alternative, we will focus on using just the most effective individual mutant operators to build second order mutants. We also plan to include **QOI** and **QOD** operators despite their weak performance, as we observed that expressing properties at the wrong level of quantification is a very common mistake, but also involves changes to the resulting subformula.

VI. Threats to Validity

The Alloy4Fun benchmark is representative of mistakes that novice users would make; thus, our results may not generalize to faults made by expert users. While μAlloy is intended to benefit all users, we believe that novice users are more likely to seek out the use of μAlloy to build confidence in their model. The Alloy4Fun benchmark is used both to find common novice mistakes as well as evaluate our new mutant operators. While there is this redundancy, our results highlight that targetting common mistakes alone is still not enough to capture all the nuances of mistakes users make, even within the same dataset.

VII. Related Work

Testing and Debugging Alloy Models. The most closely related work to μAlloy is TAR, a mutation-oriented repair technique that is aimed at repairing Ally4Fun models. TAR does have some differences in mutation operators. For instance, for a binary formula, the binary logical operators are combined together into one mutant operator group for TAR. TAR does consider higher order mutants, but this is done through applying chains mutants to a faulty location searching for a valid patch. As a result, TAR's execution is tailored to strategically generate higher and higher orders of mutants until a patch is found. In comparison, our efforts explore the viability or new and higher order mutant operators to aid in more effective mutation testing, rather than automated repair.

Besides TAR, there are several bodies of work that focus on testing and debugging Alloy models. For automated repair, ARepair is a generate and validate automated repair technique that uses AUnit test cases to outline expected behavior [27] and ICEBAR extends ARepair to additionally consider built-in Alloy assertions to guide the repair and check candidate patches [12]. In addition, ATR is an Alloy repair technique that tries to find patches based on a preset number of templates and uses Alloy assertions as an oracle. For fault localization, AlloyFL is a hybrid fault localization technique that takes a faulty Alloy model and an AUnit test suite and returns a ranked list of suspicious locations [29] and FLACK is a fault localization technique that locates faults by using a partial max sat toolset to compare the difference between a satisfying instance and counterexamples [31]

Mutation Testing. Mutation testing [9], [13] is an active research area [15] that is well studied for imperative languages but is lesser explored for declarative languages [7]. Closely related work introduces mutation testing for model checkers [3], [8], [21], but the automated analysis is different, resulting in different strategies fo executing and checking mutants. Our work is also closely related to mutation testing efforts for other specification languages. Srivatanakil et al. [23] who define mutation operators for CSP specifications written using FDR2 syntax [1]. Aichernig and Salas [2] define specification mutation for OCL and apply it to pre/post-condition specifications for constraint-based testing.

VIII. Conclusion

This paper explores trying to make Alloy's mutation operators better reflect common mistakes through new mutant operators and second order mutants. Our experimental results reveal that our new mutant operators close the gaps in fault detection, but lack the ability to accurately represent mistakes users actually make. In contrast, our second order mutants overlap significantly with real mistakes, but are not feasible due to their overhead. However, this opens the door to consider how to generate valuable subests of second order mutants.

References

[1] Software FDR2. http://www.fsel.com/software.html

[2] Aichernig, B.K., Salas, P.A.P.: Test case generation by ocl mutation and constraint solving. In: QSIC. pp. 64–71 (2005)

[3] Aichernig, B.K., Lorber, F., Ničković, D.: Time for mutants—model-based mutation testing with timed automata. In: Tests and Proofs: 7th International Conference, TAP 2013, Budapest, Hungary, June 16-20, 2013. Proceedings 7. pp. 20–38. Springer (2013)

[4] Akhawe, D., Barth, A., Lam, P.E., Mitchell, J., Song, D.: Towards a formal foundation of web security. In: 2010 23rd IEEE Computer Security Foundations Symposium. pp. 290–304 (2010)

[5] Alloy4Fun Benchmark: https://zenodo.org/record/4676413 (2022)

[6] Bagheri, H., Kang, E., Malek, S., Jackson, D.: A formal approach for detection of security flaws in the android permission system. In: Formal Aspects of Computing. p. 544 (2018)

[7] Belli, F., Jack, O.: Declarative paradigm of test coverage. Software Testing, Verification and Reliability **8**(1), 15–47 (1998). https://doi.org/10.1002/(SICI)1099-1689(199803)8:1<15::AID-STVR146>3.0.CO;2-D, http://dx.doi.org/10.1002/(SICI)1099-1689(199803)8:1<15::AID-STVR146>3.0.CO;2-D

[8] Black, P.E., Okun, V., Yesha, Y.: Mutation of model checker specifications for test generation and evaluation. Mutation testing for the new century pp. 14–20 (2001)

[9] DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on test data selection: Help for the practicing programmer. Computer **4**(11) (Apr 1978)

[10] Dini, N., Yelen, C., Alrmaih, Z., Kulkarni, A., Khurshid, S.: Korat-API: A framework to enhance Korat to better support testing and reliability techniques. In: SAC (2018)

[11] Eid, E., Day, N.A.: Static profiling alloy models. IEEE Transactions on Software Engineering pp. 1–1 (2022)

[12] Gutiérrez Brida, S., Regis, G., Zheng, G., Bagheri, H., Nguyen, T., Aguirre, N., Frias, M.: ICEBAR: Feedback-Driven Iterative Repair of Alloy Specifications. Association for Computing Machinery, New York, NY, USA (2023)

[13] Hamlet, R.G.: Testing programs with the aid of a compiler. IEEE Trans. Softw. Eng. **3**(4), 279–290 (Jul 1977). https://doi.org/10.1109/TSE.1977.231145, http://dx.doi.org/10.1109/TSE.1977.231145

[14] Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2006)

[15] Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. **37**(5), 649–678 (Sep 2011). https://doi.org/10.1109/TSE.2010.62, http://dx.doi.org/10.1109/TSE.2010.62

[16] Jovanovic, A., Sullivan, A.: Mutation testing for temporal alloy models. In: 26th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS. pp. 228–238 (2023)

[17] Jovanovic, A., Sullivan, A.: Right or wrong: Understanding how novice users write software models. arXiv preprint arXiv:2402.06624 (2024)

[18] Maoz, S., Ringert, J.O., Rumpe, B.: CD2Alloy: Class diagrams analysis using Alloy revisited. In: MODELS (2011)

[19] Marinov, D., Khurshid, S.: TestEra: A novel framework for automated testing of Java programs. In: ASE (2001)

[20] Nelson, T., Barratt, C., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: The Margrave tool for firewall analysis. In: LISA (2010)

[21] Okun, V., Black, P.E., Yesha, Y.: Testing with model checker: Insuring fault visibility. In: Proceedings of 2002 WSEAS international conference on system science, applied mathematics & computer science, and power engineering systems. pp. 1351–1356 (2003)

[22] Samimi, H., Aung, E.D., Millstein, T.D.: Falling back on executable specifications. In: ECOOP. pp. 552–576 (2010)

[23] Srivatanakul, T., Clark, J.A., Stepney, S., Polack, F.: Challenging formal specifications by mutation: A CSP security example. In: APSEC (2003)

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

[25] Sullivan, A., Zaeem, R.N., Khurshid, S., Marinov, D.: Towards a test automation framework for Alloy. In: Proceedings of the 2014 SPIN Workshop on Software Model Checking. pp. 113–116 (2014), http://doi.acm.org.ncat.idm.oclc.org/10.1145/2632362.2632369

[26] Trippel, C., Lustig, D., Martonosi, M.: Security verification via automatic hardware-aware exploit synthesis: The CheckMate approach. IEEE Micro (2019)

[27] Wang, K., Sullivan, A., Khurshid, S.: Automated model repair for Alloy. In: ASE (2018)

[28] Wang, K., Sullivan, A., Khurshid, S.: MuAlloy: A Mutation Testing Framework for Alloy. In: Proceedings of the 40th International Conference on Software Engineering (ICSE) Demo Track. pp. 29–32 (2018). https://doi.org/10.1145/3183440.3183488

[29] Wang, K., Sullivan, A., Khurshid, S.: Fault localization for declarative models in Alloy. In: ISSRE (2020)

[30] Zaeem, R.N., Khurshid, S.: Contract-based data structure repair using Alloy. In: ECOOP. pp. 577–598 (2010)

[31] Zheng, G., Nguyen, T., Gutiérrez Brida, S., Regis, G., Frias, M.F., Aguirre, N., Bagheri, H.: Flack: Counterexample-guided fault localization for alloy models. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). pp. 637–648 (2021)