Right or Wrong – Understanding How Users Write Software Models in Alloy

Ana Jovanovic and Allison Sullivan(✉)

The University of Texas at Arlington, Arlington, TX, USA
{ana.jovanovic,allison.sullivan}@uta.edu

Abstract. Writing declarative models has numerous benefits, ranging from automated reasoning and correction of design-level properties before systems are built, to automated testing and debugging of their implementations after they are built. 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. However, even with visualized scenarios, it is difficult to write correct Alloy models. This paper presents an empirical study of over 93,000 models written mainly by users trying to learn Alloy to form a comprehensive series of observations to guide the growing body of debugging and educational efforts for Alloy model development.

1 Introduction

In today's society, we are becoming increasingly dependent on software systems. However, we also constantly witness the negative impacts of buggy software. One way to help develop better software systems is to leverage software models. Alloy [17] is a relational modeling language. A key strength of Alloy is the ability to develop models in the Analyzer [2], an automatic analysis engine based on off-the-shelf SAT solvers, which the Analyzer uses to generate scenarios that highlight how the modeled properties either hold or are refuted, as desired. The user is able to iterate over these scenarios one by one, inspecting them for correctness. Alloy has been used to verify software system designs [6,11,34,36], and to perform various forms of analyses over the corresponding implementation, including deep static checking [14,18], systematic testing [24], data structure repair [35], automated debugging [15] and to generate security attacks [1,25,29].

However, to gain the many benefits that come from utilizing software models, the model itself needs to be correct. For Alloy models, there are two types of errors that can appear: (1) *underconstrained* errors in which the model allows scenarios it should prevent, and (2) *overconstrained* errors in which the model prevents scenarios it should allow. To detect an underconstrained error, during scenario enumeration, the user needs to observe a scenario that they did not expect to see. Unfortunately, even if the user notices something is wrong with a scenario, according to a recent user study, both novice and expert Alloy modelers

A. Madeira and A. Knapp (Eds.): SEFM 2024, LNCS 15280, pp. 309–327, 2025.
https://doi.org/10.1007/978-3-031-77382-2_18

struggle to refine a faulty formula given an incorrectly produced scenario [23]. To detect an overconstrained error, after enumeration, the user needs to realize that an expected scenario was never present. Since Alloy commands can produce hundreds of scenarios, this places a high burden on the user.

Given these difficulties, there is a growing body of work looking at how to debug faulty Alloy models. AUnit is a unit testing framework [27,28], which enables users to check if a specific scenario is allowed or prevented. AUnit has been used for mutation testing [20,31], fault localization [32], and automated repair [16,30]. Since then, fault localization and repair of Alloy models has become an active research area [9,10,38,39]. As this field grows, we believe it is important to understand what mistakes developers actually make when writing models so that these techniques can be effective for the majority of mistakes made.

In addition, Alloy has a reputation for being a user friendly formal method, mainly due to its IDE the Analyzer. As a result, over the years, there have been several efforts to build introductory formal methods courses centered around Alloy and the Analyzer [5,8,21,22,26]. Unfortunately, these studies have revealed that even with the Analyzer, new users still feel overwhelmed trying to learn the language. Therefore, to improve debugging and educational efforts for Alloy, this paper presents an empirical study that explores 93,283 submissions made by students learning Alloy to Alloy4Fun, an online educational website for Alloy. In this paper, we make the following contributions:

Empirical Study: We present a systematic study of models written by Alloy users with revision histories that capture back-to-back incremental changes. We explore questions spanning correct and incorrect models, including: how hard is it for users to make a model that compiles, how do user's correct answers differ from expert oracles, what common mistakes do users make, and how often are users actually close to a correct formula.

Practical Impacts: We distill our observations into practical guidelines for future work in debugging models and building educational material.

Benchmarks for Alloy: We reorganize the dataset into two benchmarks: (1) a collection of faulty models broken down by type of fault and (2) a collection of models that tracks incremental changes, which can be used to evaluate and improve incremental analysis techniques for Alloy [7,19,33,37].

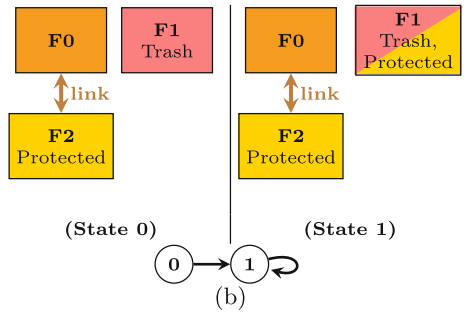
Open Source: We release our benchmarks and analysis at: https://github.com/AlloyUserStudy/Alloy4FunDataAnalysis.

2 Background

In this section, we describe the key concepts of Alloy and Alloy4Fun.

```
var sig File { var link : lone File }
var sig Trash in File {}
var sig Protected in File {}
//protected file is never sent to trash
pred prop9 { no Protected & Trash }
//SECRET
pred prop9o { always no Protected & Trash }
check prop90k { (prop9 and not prop9o)
   implies (prop9o iff prop9) }
```

(a)



(b)

Fig. 1. Faulty Submission of an Alloy Model of a File System Trash Can

2.1 Alloy

Figure 1 (a) displays a faulty temporal model of a file system trash can from the Alloy4Fun dataset [4]. Signature paragraphs introduce named sets and can define relations, which outline relationships between elements of sets. Line 1 introduces a named set `File` and establishes that each `File` atom connects to zero or one (`lone`) `File` atoms through the `link` relation. Lines 2 and 3 introduce the named sets `Trash` and `Protected` as subsets (`in`) of `File`. Signatures and relations can be declared mutable (`var`), which means that the elements of these sets can vary across different states in the same scenario. In our example, all 3 signatures and the one relation are mutable.

Predicates introduce named formulas that can be invoked elsewhere. The predicate `prop9` on line 5 uses empty set (‘`no`’) and set intersection (‘`&`’) to incorrectly attempt to establish that a protected file is never sent to the trash. However, since the signatures are mutable, the incorrect version is true as long as no protected files are in the trash for the *first* state but does not require the constraint to be true in *every* state. To correct this, the linear temporal operator `always` can be appended to the start.

2.2 Alloy4Fun Exercises

Alloy4Fun is an educational platform for Alloy where users can write and compile Alloy models through an online interface. Alloy4Fun contains several starter models in which the signature paragraphs are given. Each starter model contains a varying number of empty predicates with a corresponding English description of the property that the user can attempt to encode into formal logic. To check their answer for a given predicate, Alloy4Fun will compare if their written submission is equivalent to a hidden oracle. If their submission is not equivalent to the oracle, the user will be presented with a counterexample: a scenario that depicts a situation where the user’s formula behaves differently than the oracle. The user can then continue to iterate on their formula until it is correct.

To illustrate, in Fig. 1 (a), every line after the `SECRET` comment is hidden from the user. The check command (lines 8–9) compares if a submission is equivalent

to the oracle. Commands indicate which formulas to invoke and what scope to explore. The scope places an upper bound on the size of all signature sets and the number of state transitions. The command on line 8 instructs the Analyzer to search for an assignment to all sets in the model using up to 3 `File` atoms and up to 10 state transitions by default. Figure 1 (b) displays the first counterexample scenario found by the Analyzer, which informs the user that their model is incorrect: in state 1, `F1` is both in the trash and protected.

3 Experiment SetUp

In this paper, we will use the following terminology. A *submission* consists of the base model, the user's current attempt for a specific exercise, and any helper predicates invoked. An *edit path* is the series of submissions a user makes starting with the starter model and ending with their last submission. Edit paths can contain submissions across multiple different exercises but are all within the same base model. An *attempt* is the series of submissions within an edit path that the user makes for a specific exercise.

3.1 Experimental Data

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 [4]. In total, there are 97,755 submissions that span 17 different Alloy models and 183 predicates to be filled in (exercises). Of these, we filtered out 2,024 submissions that change the model's structure or are only changes to the theme settings for visualization. In addition, we filtered 2,448 submissions that are empty. Empty submissions are underconstrained faults. However, since empty submissions do not fundamentally illustrate any effort by the user, we removed them to avoid skewing the conclusions. As a result, we end up with 93,283 submissions, all of which have tangible user content.

Table 1 gives an overview of the complexity of the models used in our study in terms of the universe of discourse each model creates. 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. 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.

For the 183 exercises, we classify them into one of the following categories: relational logic (RL), predicate logic (PL), first-order logic (FOL), and linear temporal logic (LTL). PL and LTL contain their traditional operators as supported within Alloy's grammar [3]. RL operators contain all set theory operators,

comparison operators, and multiplicity operators from Alloy's grammar. FOL contains both quantifier operators and comprehension operators. Exercises were classified based on the highest operator present in the oracle solution according to the partial order of RL → PL → FOL → LTL. In total, 51 are RL exercises, 1 is a PL exercise, 95 are FOL exercises and 36 are LTL exercises.

Table 1. Complexity of Base Models

	#Sig	#Abs	#Ext	#Rel	Arity	#Exe	#AST
classroom_fol	5	0	2	3	2.33	15	10.00
classroom_rl	5	0	2	3	2.33	15	10.13
courses_v1	6	0	2	5	2.2	15	16.87
courses_v2	6	0	2	5	2.2	15	16.87
cv_v1	5	1	2	4	2	4	19.75
cv_v2	5	1	2	4	2	4	21.75
graphs	1	0	0	1	2	8	7.63
lts	3	0	1	1	3	6	19.71
productionLine_v1	5	0	2	3	2	4	14.25
productionLine_v2	10	1	7	4	2	10	14.90
productionLine_v3	10	1	7	4	2	10	14.90
socialMedia	5	0	2	5	2	8	15.75
trainstation_fol	7	0	5	2	2	10	13.40
trainstation_ltl	6	0	4	3	2	17	23.44
trash_rl	3	0	2	1	2	10	4.80
trash_ltl	3	0	2	1	2	20	8.55
trash_fol	3	0	2	1	2	10	4.80
AVG	**5.18**	**0.24**	**2.71**	**2.94**	**2.12**	**10.65**	**13.97**

3.2 Classification of Submissions

Within our dataset, a submission falls into one of the following categories: correct, overconstrained (over), underconstrained (under), both over- and underconstrained (both), type error (type), and syntax error (syntax). Correct submissions are semantically equivalent to the oracle, overconstrained submissions prevent valid scenarios from being generated, underconstrained submissions allow invalid scenarios to be generated, both submissions allow a combination of over- and underconstrained behaviors, type errors are formulas that fail to typecheck, and syntax errors are formulas that fail to adhere to the grammar rules. To determine which category a submission falls under, we use Alloy4Fun's hidden oracle check command for correctness. If not correct, we use the following command templates to determine the type of fault:

```
under: check {prop9 and !prop9o}
over: check {!prop9 and prop9o}
```

In addition, if the model fails to compile, we use the generated error message to determine if there was a type or a syntax error.

3.3 Recreating Submission Revision Histories

When a student checks an exercise, Alloy4Fun logs the student submission with a unique id key and a "derivationOf" parameter that contains the unique id

of the parent entry that is either the original Alloy4Fun example model or a previous submission. We use this information to re-build edit paths that capture the series of edits a user makes. As an example, the following is the edit path for our example in Fig. 1 where U means underconstrained and C means correct:

```
inv9 U: no Protected & Trash
inv9 C: always no Protected & Trash
```

For edit paths, we do include empty submissions. While empty submissions do not hold much value as individual submissions, in the middle of an edit path, an empty submission can convey when users reset their attempt on an exercise.

3.4 Methodology

We first explore general trends across all submissions (RQ1-RQ2) before exploring how users write invalid submissions (RQ3), correct submissions (RQ4), and incorrect submissions (RQ5-RQ6). Finally, we consider how users incrementally problem solve (RQ7).

For RQ1 and RQ2, we analyze submissions based on their classification. For RQ2, we additionally separate submissions into unique versus duplicate buckets. We consider two forms of equivalence: syntactic and semantic. Syntactical duplicates covey the rate at which users literally write the same exact formula, while semantic duplicates allow us to explore the rate at which users express the same underlying concept. To determine if two submissions are syntactically equivalent, we used a PrettyStringVisitor to reprint formulas in a consistent format that removes trivial differences in formatting, such as white spaces. For semantically equivalent, we used the Analyzer to check for logical equivalence.

For RQ3, we investigate attempts that have invalid (syntax and type errors) submissions. We also pull out back to back edits in attempts in which the user repeatedly makes 5 or more invalid submissions in a row, in order to manually inspect these edits to determine what aspects of the Analyzer's error reports may be preventing users from correcting the structure of their formula.

For RQ4, we consider only correct submissions. We rank correct submissions from most repeated submission to least per exercise. From there, we then manually analyze attributes of the top submitted formulas' structure and compare different AST properties to explore how user submissions differ from the oracle formula. Similarly, for RQ5, we focus on "hot hit" submissions in which the same syntactic faulty submission is made at least 10 times. As with correct submissions, we manually investigated each "hot hit" and label it with core tenants of the mistake. In addition, for all faulty submissions, we also we use MuAlloy, a mutation testing tool [20], to generate all first-order mutants. We then check if any of the mutated models are equivalent to the oracle submission. If so, we classify the submission as "one mistake away" and explore different attributes of these fixable models in RQ6.

Finally, for RQ7, we analyze the changes in classifications that happen within back to back edits for each edit path.

4 Empirical Evaluation

4.1 RQ1: What Classification of Submissions Do Users Make?

To get a general overview of the submissions made, Fig. 2 shows pie charts depicting the breakdown in the classification of submissions, ranging from all submissions to a breakdown per type of logic. Submissions were sorted into different types of logic based on the logic in the user's submission.

As the pie charts indicate, users are the most effective at writing relational logic properties, which users express correctly 37.05% of the time. In contrast, users struggle more with FOL and LTL properties, in which less than a fourth of the submissions are correct: 23.62% and 24.13% respectively. This decrease in accuracy for FOL and LTL exercises is expected, as the introduction of quantified formulas and temporal constraints is non-trivial.

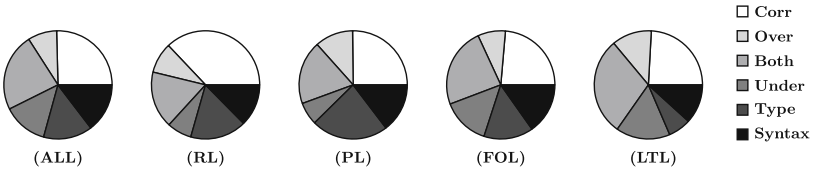


Fig. 2. Breakdown of Submission Results

While correct submissions are the largest individual chunk at 25.4%, faulty submissions (both, over and under) combine to account for 45.3% of all submissions. Faults that are both over- and underconstrained account for the largest portion of mistakes at 23.28% of all submissions. As RQ6 will highlight, this implies that faulty models are not likely to have small, simple corrections. In addition, as the complexity of the formula structure increases (FOL and LTL), users are also more likely to write properties that are accidentally too permissive. Notably, most Alloy fault localization and repair techniques struggle to perform when the issue is that constraints are missing [9,30].

Invalid submissions combine to account for 29.28% of all submissions and are almost evenly split between syntax errors (14.64%) and type errors (14.65%). While the rate of syntax errors is relatively consistent across types of logic, LTL formulas do have notably less type errors.

> **Finding 1:** For LTL formulas, only some of the signatures and relations are mutable, which narrows down for the user which elements should appear in temporal constraints. This could be the reason for LTL formula's reduction in type errors. FOL formulas could benefit from this, as educators can emphasize that the domain of the quantified formula also limits the types that should be reasoned over. In addition, if a quantified formulas trivially

reasons over sets with different types than the domain, debugging efforts should focus on modifying these sets to match the type of the domain.

4.2 RQ2: How Often Do Users Make the Same Submission?

If users often repeat the same submission, then this represents underlying common approaches that users take to modeling. Therefore, we investigate how often new users make duplicate submissions. Table 2 show the rate at which submissions are syntactically and semantically unique broken down by classification (**Clx**) and additionally type of logic for syntactic. Column #**Sub** is the total number of submissions, column #**Uni** is the number of submissions that are unique syntactically or semantically, **%Uni** is the percentage of total submissions that are unique. Syntax and Type submissions do not have semantically equivalent values since they do not compile.

Finding 2: Users do have common approaches to modeling – only 55.55% of submissions are syntactically unique and 16.55% are semantically unique. Therefore, there are lessons to be learned about these commonalities that can inturn improve education and debugging techniques.

There are two trends that impact this. First, there is high redundancy in correct submissions, which is expected as semantically, all correct submissions are equivalent. Second, there is notably less redundancy in type and syntax errors, meaning that users make a broader range of mistakes when incorrectly structuring a formula compared to writing a formula that is wrong.

Table 2. Rate of Syntactically Unique Submissions

RL						PL					
Cx	#Sub	Syntatic		Semantic		Cx	#Sub	Syntatic		Semantic	
		#Uni	%Uni	#Uni	%Uni			#Uni	%Uni	#Uni	%Uni
Correct	4418	397	15.70	51	0.56	**Correct**	390	60	13.93	1	0.36
Both	2029	1139	62.32	1061	37.29	**Both**	294	208	72.49	103	44.98
Over	1091	558	51.52	293	18.16	**Over**	177	100	44.00	22	22.00
Under	880	434	65.89	212	17.01	**Under**	105	77	80.85	9	19.15
Syntax	1503	1150	84.88			**Syntax**	227	199	78.63		
Type	2004	1460	77.07			**Type**	356	279	83.64		
SUM/AVG	11925	5138	43.09	1617	19.21	**SUM/AVG**	1549	923	59.59	135	13.98

FOL						LTL					
Cx	#Sub	Syntatic		Semantic		Cx	#Sub	Syntatic		Semantic	
		#Uni	%Uni	#Uni	%Uni			#Uni	%Uni	#Uni	%Uni
Correct	17373	3800	21.85	94	0.73	**Correct**	1513	205	12.58	30	2.06
Both	17561	11592	65.47	5946	35.33	**Both**	1831	1191	64.70	744	41.10
Over	6081	3672	61.61	1135	20.07	**Over**	748	420	53.64	209	28.67
Under	10459	5422	50.06	857	8.45	**Under**	1015	522	50.68	147	14.33
Syntax	11188	9361	82.50			**Syntax**	736	643	87.03		
Type	10878	8596	78.33			**Type**	426	330	76.64		
SUM/AVG	73540	42443	57.71	8032	15.60	**SUM/AVG**	6269	3311	52.82	1130	22.13

4.3 RQ3: How Effective Is the Analyzer's Compilation Error Reports?

Nearly a third of the time, a user will create a submission that fails to compile. Therefore, we investigate the effectiveness of Alloy's current compiler-based error reporting, by looking at the rate at which users fix these errors.

Table 3. Details for Invalid Attempts

Model	% w/ Invalid	% Nvr Valid	Subpath Len. Avg	% ≥ 5	Max
classroom_fol	30.44	2.75	1.93	7.50	11
classroom_rl	38.68	4.44	2.04	8.63	15
courses_v1	48.89	1.79	2.10	8.89	22
courses_v2	50.28	2.71	1.81	4.57	20
cv_v1	41.90	5.93	1.81	2.91	18
cv_v2	43.48	0.00	1.87	5.66	9
graphs	26.18	2.09	1.76	4.29	9
lts	40.95	8.62	2.34	10.11	17
productionLine_v1	32.91	1.28	2.10	8.77	15
productionLive_v2	41.43	1.00	1.84	5.15	37
productionLine_v3	36.38	1.54	1.84	4.65	11
socialMedia	48.27	4.19	2.06	8.56	23
trainStation_fol	50.25	0.98	1.91	6.20	11
trianStation_ltl	37.73	4.09	1.98	6.85	12
trash_fol	26.40	4.40	1.96	8.67	11
trash_ltl	32.35	0.67	1.91	7.71	9
trash_rl	28.81	3.17	2.10	8.67	15
SUM/AVG	**42.06**	**2.88**	**1.99**	**7.48**	**37**

This is summarized in Table 3. For each model, column 2 reports the percentage of attempts that have at least one invalid submission, and column 3 presents the percentage of attempts that never turned into valid formulas. The remaining three columns quantify how a user goes from an invalid submission to a valid submission. We first collected all invalid subpaths, which are the portions of attempts that consist of only type or syntax error edits. Then, we report the average length, the percentage of subpaths that have a length of 5 or more, and the max length. Compilation errors are prevalent - over 40% of attempts have them, but error reports do help users correct an error within 2 attempts on average.

Mishandling Primitive Type Errors. While our results highlight that users are often able to work through compilation mistakes, there is still the 7.88% of attempts that fall into long edit chains to try and make the model compile and the 2.88% of attempts in which the users give up. Looking into these attempts, we discovered that the Analyzer's current error reporting does not handle primitive type errors well. A primitive type error occurs when the user creates a formula that produces a set when a boolean is expected, or vice versa. To demonstrate, consider the following chain of edits for `courses_v1 inv12`:

```
inv12 T: all s:Student | s.enrolled.grades
inv12 T: all s:Student | s.enrolled.grades.s
inv12 T: all s:Student | s.(s.enrolled.grades)
```

With each edit, the user is informed that the highlighted text must "*be a formula expression.*," There is no definition of a "formula expression" and no mention of the expectation that this formula needs to evaluate to a boolean. As a result, this user keeps trying to edit the highlighted formula as if the type error is within the highlight. However, the highlighted formulas typecheck individually. In the end, the user never realizes that the issue is that the quantified formula encompassing the highlighted formulas expects a boolean not a set subformula.

> **Finding 3:** The fact that Alloy formulas inherently produce a set or a boolean result is a subtle feature of the language that is not well highlighted in Alloy's own compiler-based error reports. Since users are able to better resolve other syntax and type errors, updating error reports to directly call out the primitive type error could help prevent the long error chains, especially if the highlighted formula ambiguously compiles in isolation.

4.4 RQ4: How Do Users' Correct Answers Differ from the Oracle?

Table 4. Location of Oracle

Type	Top 1	Top 5	Top 10	Nil
RL	22	44	49	3
PL	1	1	1	0
FOL	25	58	65	24
LTL	12	21	22	9
SUM	**60**	**156**	**169**	**36**

The oracle solution often represents one of several ways in which the property could be expressed. Therefore, we wanted to explore the ways novice users' thinking may differ from the expert-written oracles. Table 4 displays the location of the oracle submission derived by ranking all syntactically unique correct submissions in decreasing order from most to least duplicate submissions. Column **Type** displays the type of exercises under consideration based on the oracle. The remaining columns show the number of exercises for that type in which the oracle is in the **Top 1** location, **Top 5**, **Top 10**, or never submitted (**Nil**).

For RL exercises, the oracle is frequently submitted by users. RL exercise use the smallest subset of operators, and this seems to translate to less diversity in how a property can be correctly expressed. This is further supported by Table 2, in which only 15.7% of correct RL submissions are syntactically unique. However, there is a notable dropoff in users creating the oracle solution for FOL and LTL exercises. By location 10, the oracle is only present in 68.42% and 61.11% of these exercises respectively. Moreover, a forth of the time, the oracle is never submitted for these exercises.

> **Finding 4:** In conjunction with Table 2, it is clear that correct FOL exercises simply have a lot of syntactic diversity. The structure of a FOL formula introduces a lot of opportunities for user's to make design choices, such as selecting the quantifier and selecting the domain, leading to wide variability and loweing the oracle's presence. However, correct LTL submissions lack diversity but also fail to utilize the structure of the oracle.

Top Submission Differences With Oracle. To build a better understanding of why LTL exercises often do not contain the oracle submission, and to further investigate the differences users make compared to the oracle, we manually investigated the differences between the top 1 correct solution and the oracle. Table 5

displays for each identified difference the number of top 1 correct submissions that have this difference (column # **Occ**) and the percentage that this accounts for out of all top 1 submissions (column **Rate**).

Table 5. Differences - Oracle and Top 1

Difference	# Occ	Rate
Upscaled quantification	55	29.25
Downscaled quantification	21	11.17
Different quantification domains	11	5.85
Sim, Different order of operands	12	6.38
Sim, Formed different sets	7	3.72
Sim, Different operator	19	10.11
Sim, Additional trivial operator	1	0.53
Expanded out formula	2	1.06
No Change	60	31.91

35 submissions are similar (sim) but involve minor differences, such swapping the order of operands. Many of these submissions are LTL exercises in which the location of the temporal operator `always` varies. In addition, the three of the four "additional trivial operator" are LTL exercises in which the user appends an unnecessary leading `always` operator. Together, these similar submissions highlight common modeling preferences that result in the oracle being less present for LTL exercises.

> **Finding 5:** Users prefer leading with temporal operators, while the oracle solutions insert them close to the subformula of interest. For instance, a user will write "`always all a: A | F`" while the oracle will use "`all a : A | always F`." However, this preference does not always create equivalent formulas. Debugging techniques should consider patch steps that relocate temporal operators closer to the subformula of interest.

Size of Correct Solutions. Overall, the most prevalent difference in Table 5 is that users increase the degree of quantification in the formula, which is seen in 55 of the top 1 solutions. For instance, consider the following for `inv1` from the `productionLine_v2`. Only 21 correct submissions match the oracle (o) (8.71%) while 101 correct submissions (41.91%) use the quantified formula (s).

```
o: Worker = Human + Robot //  Workers are either human or robots
s: all w: Worker | w in Human or w in Robot
```

To quantify this impact, Table 6 displays the Abstract Syntax Tree (AST) difference between the oracle solution and all of the correct submissions. Column $\mathbf{Dif}_{Tot}$ displays the average difference in total number of AST nodes between the oracle and all correct submissions. The next four columns display the average difference in total number of AST nodes of each type of logic. For these columns, a negative number means the oracle had less AST nodes. To give context to how significant the difference in nodes is, column $\mathbf{Dif}_{Mag}$ displays average magnitude in difference between the oracle and correct submissions.

RL and PL exercises see the largest increase in size, with the oracle being on average a 0.38× smaller formula. This is not surprising, as users writing FOL versions instead would naturally increase the size. However, even FOL exercises, which already have a quantified formula, also see a large increase in the number

of AST nodes. Therefore, even when quantification is expected, users add more nested quantification than the oracle formulation.

> **Finding 6:** Users are not likely to rely on condensed relational logic to express properties, and will often write verbose formulas that apply first order logic more than is actually needed to express the constraint. Since this can impede execution runtime, it is worthwhile investing in discovering refactoring strategies to reduce quantification.

Table 6. AST Differences Between Oracle and Corr. Subs

Type	Dif_{Tot}	Dif_{RL}	Dif_{PL}	Dif_{FOL}	Dif_{LTL}	Dif_{Mag}
RL	−11.53	−2.49	−0.69	−1.16	−0.02	0.38
PL	−10.74	−1.79	−0.03	−1.26	0.00	0.38
FOL	−8.17	−1.87	−0.82	−0.52	0.00	0.81
LTL	−3.70	−0.89	−0.43	−0.24	−0.07	0.77
AVG	**−9.08**	**−2.03**	**−0.76**	**−0.72**	**−0.01**	**0.66**

4.5 RQ5: What Mistakes Do Users Make When Writing Formulas?

As indicated by Table 2, users often repeat the same mistake. To learn what these mistakes are, we manually investigate and label each "hot hit" faulty submission with the main aspects of the mistake. Correspondingly, Table 7 displays the common mistakes made broken down by classification of submission and type of logic. Column #**Occ** represent the number of unique "hot hit" submissions of that mistake and column #**Sub** represents the total number of submissions with duplicates included. In green, we highlight the top 4 mistakes.

Across all logic types, the single most common mistake is the incorrect use of an operator. Our dataset includes a detailed breakdown of which operators are incorrectly applied and at what rates. Expectedly, several of these mistakes involve users making the wrong choice between two similar operators, i.e. transitive closure ('`^`') instead of reflexive transitive closure ('`*`'). However, there were also several mistakes that reflect users picking operators that are not within the same grammar division, but are used to capture two different but similar ideas. To illustrate, consider the following submission (s) to `inv6` from `productionLine_v2` that 20 users submitted and the closest correct solution (c):

```
// Components built of dangerous parts are also dangerous
s: all c: Component | c.parts in Dangerous => c in Dangerous
c: all c: Component | some c.parts & Dangerous => c in Dangerous
```

As shown, several users choose to encode this by checking some set intersection instead of a subset check. For their formulas, these two operators behave the

Table 7. Common Mistakes Made by Users (subs with at least 10 repeats)

Difference	RL		FOL		LTL		Total	
	#Occ	#Sub	#Occ	#Sub	#Occ	#Sub	#Occ	#Sub
More complex quant. domain	0	0	4	64	0	0	4	64
Narrowed quant. domain	0	0	**70**	**1338**	0	0	**70**	**1338**
Different quant. domain	0	0	6	183	0	0	6	183
Upscaled quant. level trying to state relationships	0	0	**101**	**1990**	**8**	**159**	**109**	**2149**
Incorrect order of nested quant.	0	0	4	78	0	0	4	78
Nested quant. disjoint mistake	0	0	17	413	0	0	17	413
Downscaled quant. level leading to incorr. expr range	**23**	**378**	15	264	2	34	40	676
Tried to inverted concept	2	37	6	144	2	38	10	219
Trying to explictly outline steps	0	0	0	0	2	24	2	24
Incorrect use of operator	**14**	**238**	**130**	**2438**	**12**	**208**	**156**	**2884**
Operator not commumtative	1	14	16	294	3	71	20	379
Missing operator	1	10	7	175	**8**	**146**	16	331
Incorrectly scoped expr, missing parentheses	0	0	1	23	0	0	1	23
Incorrect application of extension signature	3	46	**65**	**1181**	1	23	**69**	**1250**
Subportion of total concept	**5**	**128**	23	492	3	46	31	666
Wrong understanding of exercise	**4**	**61**	3	55	**3**	**53**	10	169
Total	53	912	468	9132	44	802	565	10846

same except for when the left-hand side of the operator is empty. In this case, if `c.parts` is empty, then `c.parts in Dangerous` evaluates to `true`, which prevents scenarios from being generated in which there are no dangerous components.

> **Finding 7:** While users make the wrong choice between two closely related operators, it is also clear that users make mistakes by encoding incorrect but closely related *concepts*. Both of these modeling mistakes often show up as a subtle bug in which only a few scenarios, usually involving corner cases like an empty set, differentiate the formulas. Users should explore corner case scenarios to ensure the proper boundary behavior of their constraints.

FOL submissions are prevalent in Table 7, accounting for 468 of the 565 hot hit submissions. In addition, half of the frequently repeated classification of mistakes (column **Mistakes**) involve errors related to quantification. Therefore, there are fundamental misunderstandings users have about how and when to use quantification. The most common quantifier mistake is that users try to upscale the level of quantification in an attempt to explicitly outline relationships, where upscale means the submission inserted more quantifiers than the oracle.

As an example, consider the following submission for `classroom_rl inv5`:

```
s: all t:Teacher| some t.Teaches
c: some Teacher.Teaches //There are classes assigned to teachers.
```

that accounts for 30 (35.29%) of the both over- and underconstrained submissions. In this case, the subformula is correct, but the quantifier is wrong. While there are often ways to express a RL formula with an equivalent FOL formula, the increase in complexity of the formula structure introduces more opportunities for users to make a mistake. Hand in hand with upscaling, downscaling the level of quantification is also a common mistake that often results in a constraint being satisfied incorrectly if at least one element of the set satisfies it.

Finding 8: Debugging techniques should consider transformations that change the degree of quantification, knowing that users frequently pick the wrong quantification level. Moreover, educators should emphasize when the use of quantification is appropriate to express the relationship per *atom* versus when not to use quantification to express the relationship per *type*.

Users also make mistakes picking the right domain to reason over, often by making the domain of a quantified formula too narrow. A narrower quantification domain almost always occurred on models that had extension signatures and the user used an extension signature for the domain in place of the base signature. As an example, consider the following submission for `inv3` for model `courses_v2` which accounts for 65 (40.37%) of the underconstrained submissions:

```
s: all c : Course | some p : Professor | c in p.teaches
c: all c : Course | some p : Person | c in p.teaches // Courses must have teachers.
```

Finding 9: Users can write the right subformula but use the wrong domain in quantified formulas. Debugging techniques should explore changes to the domain, with a focus on making the domain more restrictive.

4.6 RQ6: How Often Are Users "One" Mistake Away?

Our results in RQ5 indicate that users often make mistakes by encoding a similar but different concepts. However, similar concepts can be represented with notably different logic. Therefore, we wanted to explore how often users are actually close to the correct solution using MuAlloy to find submissions that are "one mistake away." Table 8 depicts the frequency at which mutants fix submissions broken down by the type of logic present in the submission and the classification of the submission. Column #**Fixes** displays the total number of mutants that fixed a submission, column # $\mathbf{Sub}_f$ shows the number of submissions that were fixed and column % $\mathbf{Sub}_f$ shows what percentage of submissions # $\mathbf{Sub}_f$ is. The number of fixes is more than the number of submissions fixed because some submissions could be fixed by multiple different mutations.

Table 8. Frequency of Mutant Corrections

Clx/Type	# Fixes	# $\mathbf{Sub}_f$	% $\mathbf{Sub}_f$
Both	1089	882	6.24
Over	1456	871	18.34
Under	1527	999	15.48
RL	247	192	5.61
PL	47	29	11.69
FOL	3092	2120	10.83
LTL	686	411	19.74
SUM	**4072**	**2752**	**10.86**

Overall, only 10.86% of submissions can be corrected by the current mutant operators. The rate is notably lowest for RL submissions, but is nearly quadrupled for LTL submissions. One reason for this is that mutant operators not do tend to create fundamentally new constraints, even

when insertion operators are applied. While LTL submissions can often be fixed with inserting or deleting temporal operators, RL submissions often need to fundamentally change the sets being formed. Case in point, only 6.24% of both over- and underconstrained faults can be fixed with mutants. This is expected as these formulas are sometimes too permissive and sometimes too restrictive, which is likely to lead to multi-step edits not captured by first-order mutants.

> **Finding 10:** The overall low fix rate is in line with observations from RQ5 in which users often make small encoding mistakes, but the difference in logic is often multiple edit steps away, not one. For instance, there is no first order mutation operator that transforms "`a in b`," into "`some a & b`." While blanket generation of second-order mutants would have scalability issues, the results from RQ5 should be utilized to produce a select subset of higher order mutants that reflects common mistakes.

Table 9. Breakdown of Mutant Operators That Fixed Subs

Mutant Op	# Fixes	Percent
Unary Operator Insertion	1269	31.16
Unary Operator Deletion	160	3.93
Unary Operator Replacement	340	8.35
Binary Operator Deletion	272	6.68
Binary Operator Replacement	810	19.89
Binary Operator Exchange	167	4.10
List Operator Deletion	221	5.43
List Operator Replacement	53	1.30
Quantifier Operator Replacement	724	17.78
Prime Operator Insertion	55	1.35

Although the fix rate is low, we still want to learn about the mutant operators that do correct models. Table 9 displays the rate at which individual mutant operators fixed submissions. Column #**Fixes** is the number of faulty submissions fixed and **Percentage** is what percentage of fixed submissions the preceding #**Fixes** represents. A majority of the fixes come from replacement (47.32%) and insertions (32.51%). The high rate of replacement fixes is supported by observations from our common mistakes (RQ5), where we found that the most frequent mistake is that users incorrectly select between similar but different operators. For insertions, Unary Operator Insertion (UOI) fixes are dominant. UOI fixes for LTL formulas almost always involve inserting temporal operators later in a formula, which is not unexpected as RQ4 highlights that users tend to push temporal operators to the front of formulas.

> **Finding 11:** Mutant operators that aligned with common mistakes are correlated with the ability to fix more faulty submissions. Therefore, it is worth investing in refining existing and creating new mutant operators based the patterns in RQ5.

4.7 RQ7: How Do Users Respond to Their Mistakes?

As users incrementally work towards a correct solution, users make changes that may move their submission from one classification to the next. Table 10 displays the classification change, or lack thereof, produced by back to back edits. The

letter `E` represents empty submissions. Row # **Edit '?'** represents the number of edits that start as one classification and move to a new one based on the corresponding column, while Row **% Edit '?'** represents what percentage this number is out of the total edits that start as classification '?'. For instance, there are 4,133 edits in which the user starts with an overconstrained formula and produces an overconstrained formula (`O-O`).

Table 10. Details of Back-to-Back Edits

Change	O-O	O-U	O-B	O-S	O-T	O-C	O-E
# Edits O	4133	526	1484	762	877	2339	84
% Edits O	40.50	5.15	14.54	7.47	8.59	22.92	0.82
Change	U-O	U-U	U-B	U-S	U-T	U-C	U-E
# Edits U	548	9472	2884	1464	1248	2603	110
% Edits U	2.99	51.68	15.73	7.99	6.81	14.20	0.60
Change	B-O	B-U	B-B	B-S	B-T	B-C	B-E
# Edits B	1918	3952	19261	2871	2545	2855	178
% Edits B	5.71	11.77	57.36	8.55	7.58	8.50	0.53
Change	S-O	S-U	S-B	S-S	S-T	S-C	S-E
# Edits S	1214	2116	3731	8371	2239	2814	203
% Edits S	5.87	10.23	18.03	40.46	10.82	13.60	0.98
Change	T-O	T-U	T-B	T-S	T-T	T-C	T-E
# Edits T	1116	1647	3233	1720	10528	2402	184
% Edits T	5.36	7.91	15.52	8.26	50.54	11.53	0.88
Change	C-O	C-U	C-B	C-S	C-T	C-C	C-E
# Edits C	692	752	676	526	659	11395	51
% Edits C	4.69	5.10	4.58	3.57	4.47	77.25	0.35
Change	E-O	E-U	E-B	E-S	E-T	E-C	E-E
# Edits E	111	181	225	261	202	344	581
% Edits E	5.83	9.50	11.81	13.70	10.60	18.06	30.50

Based on Table 10, a user is significantly more likely to make an edit that does not change the classification, which implies that users often make small edits. In addition, users do frequently go from an overconstrained to a correct formula (22.92%), which is in line with our "one step away" results in which overconstrained formulas were often close to a correct model. In contrast, syntax errors, type errors and both submissions are the most likely to lead to a student resetting their attempt by erasing down to an empty submission, which indicates for these categories users are more likely to feel as if they ended up far off from the solution. Interestingly, users do go from a correct submission to all the other classifications. There are two main reasons for this. First, some of the exercises ask for an answer using a specific type of logic, but Alloy4Fun does not restrict the available operators. Therefore, users will create a correct submission and attempt to re-write it into the required logic. Second, users refine their submission, often trying to create a more condense formula.

> **Finding 12:** Given the active research around automated repair, repair techniques should consider that users are not likely to buy into a large change to their formula. However, given RQ6 results, repair patches are likely to need to make multiple changes. Therefore, repair tools should focus on providing users with supporting evidence of the validity of larger patches, such as counterexamples with explanation templates.

5 Threats to Validity

There exist several threats to the validity of our results. First, the dataset we use does not contain any information related to what users were thinking. Therefore, we are limited in some of the conclusions we can form. Second, the users are all master's students from the University of Minho (UM) and the University of Porto (UP) between 2019 and 2023. While there is breadth to the date range of the study, we do not know the details of the educational performance and

background of the master's students, which may lack diversity. Third, MuAlloy, and the Analyzer itself, do not support higher-order quantification. In theory, a higher order formula could be correct. However, since the current tools cannot evaluate them, we do not factor them into our one step away calculations. Lastly, students could have collaborated together, resulting in higher duplication rates.

6 Related Work

Alloy4Fun. The creators of the Alloy4Fun dataset have published an experience paper outlining the first semester that they used Alloy4Fun in their classroom [22]. This paper focuses largely on lessons learned utilizing Alloy4Fun in the classroom but does contain some preliminary analysis of the data collected, such as the rate of correct versus incorrect submissions and commonly reported error messages. This data is from an early benchmark and consists of 9 models and only 5000 executions. Our analysis is more in-depth and spans the significantly larger version of the dataset.

User Studies and Empirical Studies Over Alloy. There have been a few user studies exploring how developers work with Alloy [12,13,23]. The most recent study explored the debugging behavior of novice and expert users and discovered that users struggle to refine Alloy predicates using only the visual representation of scenarios [23]. Another recent study found that if users are shown a small collection of valid and invalid scenarios before writing a predicate, then users better understand what behavior the user should be trying to codify into a constraint [13]. In addition, there was a user study that explored how users interact with different enumeration strategies [12]. While our efforts focus on how to better teach and debug constraints, these user studies highlight best practices for how to present and guide users through the output of these constraints.

Tangent to our work, there is a static profile of 1,652 publicly available Alloy models pulled from GitHub that investigates how often users engage with different parts of Alloy's grammar and explores the average size and complexity of Alloy models. This study is not concerned with the accuracy of the constraints, but does give suggestions for education based on frequently used and underutilized features. These recommendations are complementary to our recommendations.

7 Conclusion

Given Alloy's popularity and well supported toolset, there is a growing body of work to debug faulty models and to use the Alloy in an educational setting. To help guide future research directions for both of these avenues, this paper explores a dataset of over 93,000 submissions made by new users. We highlight several key findings, including common patterns in correct and incorrect submissions and the realistic effectiveness of existing mutant operators.

References

1. 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)
2. Alloy analyzer Website (2019). http://alloytools.org
3. Alloy Grammar Reference (2024). https://alloytools.org/spec.html
4. Alloy4Fun Benchmark (2022). https://zenodo.org/record/4676413
5. Askarpour, M., Bersani, M.M.: Teaching formal methods: an experience report. In: Bruel, J.-M., Capozucca, A., Mazzara, M., Meyer, B., Naumchev, A., Sadovykh, A. (eds.) FISEE 2019. LNCS, vol. 12271, pp. 3–18. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-57663-9_1
6. Bagheri, H., Kang, E., Malek, S., Jackson, D.: A formal approach for detection of security flaws in the Android permission system. Formal Asp. Comput. (2018)
7. Bagheri, H., Malek, S.: Titanium: efficient analysis of evolving alloy specifications. In: Proceedings of the 2016 24th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE), pp. 27–38 (2016)
8. Boyatt, R., Sinclair, J.: Experiences of teaching a lightweight formal method (2008)
9. Brida, S.G., et al.: Bounded exhaustive search of alloy specification repairs. In: ICSE (2021)
10. Cerqueira, J., Cunha, A., Macedo, N.: Timely specification repair for alloy 6. In: Schlingloff, B.H., Chai, M. (eds.) SEFM 2022. LNCS, vol. 13550, pp. 288–303. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-17108-6_18
11. Chong, N., Sorensen, T., Wickerson, J.: The semantics of transactions and weak memory in x86, Power, ARM, and C++. SIGPLAN Not. **53**(4), 211–225 (2018)
12. Danas, N., Nelson, T., Harrison, L., Krishnamurthi, S., Dougherty, D.J.: User studies of principled model finder output. In: Cimatti, A., Sirjani, M. (eds.) SEFM 2017. LNCS, vol. 10469, pp. 168–184. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66197-1_11
13. Dyer, T., Nelson, T., Fisler, K., Krishnamurthi, S.: Applying cognitive principles to model-finding output: the positive value of negative information. Proc. ACM Program. Lang. **6**(OOPSLA1), 1–29 (2022)
14. Galeotti, J.P., Rosner, N., Pombo, C.G.L., Frias, M.F.: TACO: efficient SAT-based bounded verification using symmetry breaking and tight bounds. TSE (2013)
15. Gopinath, D., Malik, M.Z., Khurshid, S.: Specification-based program repair using SAT. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 173–188. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19835-9_15
16. Gutiérrez Brida, S., et al.: ICEBAR: feedback-driven iterative repair of alloy specifications. Association for Computing Machinery, New York, NY, USA (2023)
17. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2006)
18. Jackson, D., Vaziri, M.: Finding bugs with a constraint solver. In: ISSTA, August 2000
19. Jovanovic, A., Sullivan, A.: Reach: refining alloy scenarios by size (tools and artifact track). In: 2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE), pp. 229–238 (2022)
20. 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)

21. Krishnamurthi, S., Nelson, T.: The human in formal methods. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) FM 2019. LNCS, vol. 11800, pp. 3–10. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30942-8_1
22. Macedo, N., et al.: Experiences on teaching alloy with an automated assessment platform. In: Raschke, A., Méry, D., Houdek, F. (eds.) ABZ 2020. LNCS, vol. 12071, pp. 61–77. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-48077-6_5
23. Mansoor, N., Bagheri, H., Kang, E., Sharif., B.: An empirical study assessing software modeling in alloy. In: International Conference on Formal Methods in Software Engineering, pp. 44–54 (2023)
24. Marinov, D., Khurshid, S.: TestEra: a novel framework for automated testing of Java programs. In: ASE (2001)
25. Nelson, T., Barratt, C., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: The Margrave tool for firewall analysis. In: LISA (2010)
26. Simonot, M.: Teaching abstraction in mathematics and computer science - a computer-supported approach with alloy (2012)
27. Sullivan, A., Wang, K., Zaeem, R.N., Khurshid, S.: Automated test generation and mutation testing for alloy. In: ICST (2017)
28. 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
29. Trippel, C., Lustig, D., Martonosi, M.: Security verification via automatic hardware-aware exploit synthesis: The CheckMate Approach. IEEE Micro (2019)
30. Wang, K., Sullivan, A., Khurshid, S.: Automated model repair for Alloy. In: ASE (2018)
31. 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
32. Wang, K., Sullivan, A., Khurshid, S.: Fault localization for declarative models in Alloy. In: ISSRE (2020)
33. Wang, W., Wang, K., Gligoric, M., Khurshid, S.: Incremental analysis of evolving alloy models. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11427, pp. 174–191. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17462-0_10
34. Wickerson, J., Batty, M., Sorensen, T., Constantinides, G.A.: Automatically comparing memory consistency models. In: POPL (2017)
35. Nokhbeh Zaeem, R., Khurshid, S.: Contract-based data structure repair using Alloy. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 577–598. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14107-2_27
36. Zave, P.: How to make Chord correct (using a stable base). CoRR abs/1502.06461 (2015)
37. Zheng, G., Bagheri, H., Rothermel, G., Wang, J.: Platinum: reusing constraint solutions in bounded analysis of relational logic. In: International Conference on Fundamental Approaches to Software Engineering (2020)
38. Zheng, G., et al.: ATR: template-based repair for alloy specifications. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2022. pp. 666–677 (2022)
39. Zheng, G., et al.: Flack: counterexample-guided fault localization for alloy models. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 637–648 (2021)