Mutation Testing for Temporal Alloy Models

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

Allison Sullivan
University of Texas at Arlington
Arlington, TX USA
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. 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. To further add to the complexity, Alloy's grammar was recently expanded to support linear temporal logic, increasing both the expressibility of Alloy as well as the burden for accurately expressing properties. To address this, this paper presents μAlloyτ, an extension to Alloy's mutation testing framework that accounts for the newly introduced temporal logic, including updating μAlloyτ's test generation capability to produce temporal test cases. Experimental results reveal μAlloyτ is efficient at generating and checking mutations and μAlloyτ's automatically generated tests are effective at detecting faulty temporal models.**

***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 [17] 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 lets users explore behavior enabled by their models. To achieve this, the Analyzer 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. As output, the Analyzer produces a collection of scenarios the user can explore. Alloy models and their corresponding scenarios have been used to validate software designs [21], [23], to test and debug code [14], [22], to repair program states [25], [34] and to synthesize security attacks [29], [4], [6].

Unfortunately, the model itself needs to be correct to gain the many benefits that arise from utilizing software models. 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 [28]. In the context of Alloy's declarative execution, in which there is no notion of imperative control flow, and the SAT solver finds all satisfying scenarios in one execution, AUnit defines: (1) what is a test case, (2) how is a test case executed and its pass/fail outcome resolved and (3) what are different types of coverage criteria. With the existence of AUnit, several traditional imperative testing practices were ported to Alloy, including mutation testing, fault localization, and repair [30], [31], [32].

μAlloy is the mutation testing framework, which generates mutants, generates a mutant-killing test suite, and performs traditional mutation testing. To generate mutants, μAlloy first defines a series of mutation operators for Alloy's first-order relational logic 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.

μAlloy is developed for Alloy 5, which is prior to the recent incorporation of linear temporal logic into Alloy [10]. As part of providing support for linear temporal logic, Alloy's structural constraints were updated to allow users to specify mutable sets of the model that can change between states. As a result, Alloy scenarios were also updated to explicitly outline multiple states rather than a single state. These changes increase the broader applicability of Alloy, which can now easily reason over the dynamic behavior of a system in addition to the structural behavior of a system. However, AUnit, and the various testing frameworks that utilize AUnit, were not designed to handle temporal constraints and state-based scenarios.

This paper introduces μAlloyτ, an expansion of the μAlloy framework that updates both AUnit and μAlloy to handle temporal logic. Importantly, μAlloy is used as a sub-process in both the fault localization (AlloyFL) and repair (ARepair) techniques that also utilize AUnit test cases. Therefore, extending μAlloy's framework to support linear temporal logic

979-8-3503-2480-8/23/$31.00 ©2023 IEEE
DOI 10.1109/MODELS58315.2023.00013

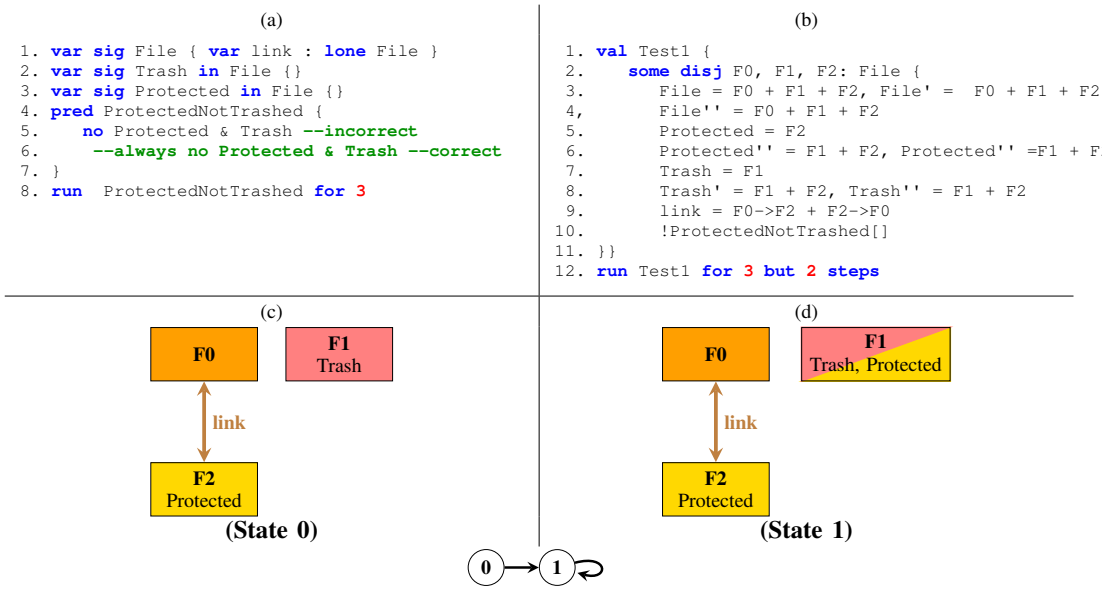


Fig. 1. Faulty Alloy Model of a File System Trash Can and Fault Revealing AUnit Test Case

opens the door to updating AlloyFL and ARepair.

In this paper, we make the following contributions:

Temporal Mutant Operators: We create mutation operators for temporal logic and dynamic signatures and relations.

Temporal Test Generation: We define test cases that reason over multiple states and provide a translation for turning state-based counterexamples into tests.

Evaluation: We evaluate our mutation testing framework and the quality of test suites produced by our test generation strategy using a benchmark of temporal Alloy models.

Open Source: We release an implementation of our framework that is built on top of version 6.0.0 of the Analyzer at https://github.com/MuAlloyT/mualloy_temporal.

II. Background

In this section, we describe key concepts of Alloy, AUnit, and μAlloy.

A. Alloy and AUnit

Figure 1 (a) displays a faulty temporal model of a file system trash can from the Alloy4Fun benchmark, which is comprised of real-world faulty models from new Alloy users [5]. 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 (`File`, `Trash`, and `Protected`) and the one relation (`link`) are mutable.

Predicate paragraphs introduce named first-order, linear temporal logic formulas that can be invoked elsewhere. The predicate `ProtectedNotTrashed` 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 of the predicate. 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) - (d) shows a fault revealing AUnit test case depicted textually (b) and graphically (c) - (d). An AUnit test case has two components: (1) a valuation, which outlines a scenario, and (2) a command, which outlines the formulas under test. A test case passes if the valuation is a valid instance for the command. Therefore, AUnit test cases allow the user to explicitly check if a specific scenario is correctly generated or prevented by the formulas under test. The graphical depiction is what the Analyzer would visually present to the user if the textual valuation in Figure 1 (b) is executed.

For the textual representation, lines 2 - 9 outline the valuation using existential quantification (`some`) and the disjoint operator (`disj`) to define unique elements used to assign values to the sets of the model (`File`, `link`, `Trash`, and `Protected`). To convey changes to mutable set of the model, the prime operator (‘`'`’) is used The prime operator allows a user to refer to future states of an expression, e.g. `e'` refers to the value of `e` shifted by one state further down the timeline. Line 10 outlines the command, which specifies that the formula under test is the predicate `ProtectedNotTrashed`

TABLE I
ALLOY 5 MUTANT OPERATORS.

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

and that the valuation is expected to be invalid ('`!`'). Line 12 is separately an Alloy command to execute the test case. In Alloy, state-based scenarios are all lassos, meaning that the last state either loops back to a previous state or loops to itself. For our example test case, the loop state is a self-loop on state 1, as seen in the timeline of states at the bottom of Figure 1. To ensure only these explicitly outlined lasso is possible, the execution command uses a scope of "`but 2 steps`."

B. μAlloy

μAlloy introduces mutation testing for version 5.0 and earlier of Alloy [27], [31], which focuses on relational, first-order logic and set theory.

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`." Separately, **QORU** replaces a quantified formula with a set multiplicity formula that physically shares the same operator spelling, e.g. "`no a : A | B`" becomes "`no A`." 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. This concretely means that to kill a mutant; the test suite needs to contain a test case that is satisfiable over one model and unsatisfiable over the other. μAlloy has a few runtime optimizations, such as stopping once a test case is able to distinguish the difference between the mutant and the original model.

C. Impact of μAlloy

Since μAlloy creates tests to kill non-equivalent mutants as you generate the mutants, mutation testing is used largely for its automated test generation capabilities. The process of encoding a scenario to spot check a predicate has been an ad hoc testing practice in Alloy for over a decade. Mutation testing simply gave the user a way to get a nice variety of these "spot checks" automatically. In addition, mutation testing also plays an active role in several of the automated repair techniques for Alloy. While most repair techniques suffer from scalability issues that prevent them from being used in practice, mutation testing is the core behind the only repair technique that currently scales [35]. The technique makes a tradeoff in robustness in order to give quick feedback that is intended to serve as guidance over providing a guaranteed patch. This is currently used in an educational setting to make suggestions to users developing models in Alloy4Fun [20].

III. TECHNIQUE

This section describes the updates made to μAlloy to provide mutation testing for temporal models. We outline the changes to mutation operators, how we create new equivalence checks for variable mutant operators, and how we update AUnit test cases to handle state-based scenarios.

A. Mutation Generation

To support linear temporal logic, Alloy's grammar was extended to support the temporal operators shown in Figure 2 in green. To account for these changes, μAlloyτ adds new mutation operators and updates several existing operators, as outlined in Table II. First, μAlloyτ establishes 4 new mutant

TABLE II
NEW AND UPDATED MUTANT OPERATORS FOR ALLOY 6

New Ops	Description
VOI	Variable Operator Insertion
VOD	Variable Operator Delection
POI	Prime Operator Insertion
POD	Prime Operator Deletion
Updated Ops	**Description**
UOR	Unary Operator Replacement
BOR	Binary Operator Replacement
UOI	Unary Operator Insertion
BOD	Unary Operator Insertion
UOD	Unary Operator Deletion
BOE	Binary Operand Exchange

operators. **VOI** inserts the `var` keyword on any originally non-mutable signatures and relations. Likewise, **VOD** deletes the `var` keyword for any originally mutable signature(s) and relation(s). These two mutant operators effectively flip whether a given signature or relation is allowed to change values between states in the same scenario. Second, **POI** and **POD** insert and delete the prime operator (`'`). As an optimization, we only look to apply **POI** mutant operators on any expression encountered that is mutable, as applying the prime operator to a static expression will always produce an equivalent mutant.

Second, μAlloyτ updates 6 existing mutant operators related to unary and binary expressions and formulas, which account for temporal operators that fit the mold of previously existing grammar structures from Alloy 5 and earlier. Some of these updates are straightforward, where the mutant operator only needs to be made aware of the new temporal operators, e.g. **UOR** is made aware that `always`, `eventually`, `after`, `before`, `historically`, and `since` form a temporal operator group and when one is encountered, **UOR** should create 5 mutants, replacing the original unary temporal operator with all of the remaining five temporal operators.

For other mutant operators, the updates were more involved as a broader context is needed to determine when to apply the mutant operator to a relevant AST node, e.g. **UOI** looks to insert the 6 unary temporal operators around any formula encountered, which is any Alloy constraint that evaluates to true or false. Alloy constraints can alternatively be expressions that evaluate to sets. In that case, **UOI** looks to insert the multiplicity unary operators (`no`, `one`, `lone`, `some`) but does not insert the temporal operators. This is because the temporal operators will only compile when inserted on a formula, while the multiplicity operators will only compile when inserted on an expression. While we do preemptively head off some situations in which creating a mutant will always result in a compilation error, for all mutants generated, μAlloyτ does check if the mutant compiles. If not, the mutant is discarded.

B. Equivalence Checking

In addition to ensuring the mutant compiles, as mentioned in Section II-B1, μAlloyτ follows μAlloy workflow to check if the mutated model is logically equivalent to the original model. While detecting equivalent mutants is a hard problem to solve in imperative languages, for most mutants, we can check if a mutated formula is equivalent to the original formula using the bi-conditional operator. For example, consider the following **UOI** generated mutant, which inserts the temporal operator `always` due to encountering the unary formula "`no Protected & Trash`" node in the AST:

```
pred MUTATED { always no Protected & Trash }
```

μAlloyτ will produce the following assertion checks for equivalence:

```
equiv:
check { ProtectedNeverTrashed[] <=> MUTATED[] }
```

This check will produce a counterexample that is equivalent to the AUnit test case outlined in Figure 1 (b) - (d). As a result, μAlloyτ will determine that this **UOI** mutant is non-equivalent and keep the mutant.

Two of the new temporal mutant operators, Variable Operator Insertion (**VOI**) and Variable Operator Deletion (**VOD**), require a modified process to check equivalence, as the operators are applied to the signature paragraphs of a model and cannot directly be checked using the biconditional operator. To check for equivalence for **VOI** and **VOD**, μAlloyτ asks the Analyzer to produce a scenario in which the mutated signature or relation changes between at least two states, as this will always kill a **VOI** or **VOD** mutant. For example, consider the following **VOD** generated mutant:

```
sig File { var link: lone File }
```

In the mutated model, the `File` signature can no longer vary its assigned values between states of a scenario. To check if this mutant is equivalent, μAlloyτ will create an intermediate model in which the signature `File` is *mutatable*. Then, μAlloyτ will pass the following command to the Analyzer over the intermediate model:

```
equiv: check { File = File' }
```

If a counterexample is found, then the **VOD** mutant is non-equivalent, as the counterexample will contain different assignments to `File` for at least two states, which will only be possible for the version of the model that contains a mutable `File` signature. Given the **VOI** or **VOD** operator, this will either be the mutant model or the original model but not both. This check also lets us catch if a **VOI** or **VOD** operator happens to be equivalent, which could be possible if, for instance, there are some additional constraints in the model that end up restricting the behavior of a mutable signature or relation such that it effectively cannot change.

C. Temporal Test Generation

When the equivalence check returns a counterexample, not only is the mutant non-equivalent, the counterexample generated can be turned into a mutant-killing test case. As a result, μAlloyτ is able to simultaneously generate mutants and a test suite that will achieve a 100% mutation score. Prior to μAlloyτ, AUnit test cases only reasoned over a static scenario. With Alloy 6, scenarios can now natively be stateful, which

```
  sigDecl ::= [var] [abstract] [mult] sig name,+ [sigExt]  fieldDecl,*  [block]
fieldDecl ::= [var] decl
     expr ::= ... expr '
     unOp ::= ! | not | no | mult | set | # | ~ | * | ^ | always | eventually | after | before | historically | once
    binOp ::= || | or | && | and | <=> | iff | => | implies | & | + | - | ++ | <: | :> | . | until | releases | since
                | triggered | ;
```

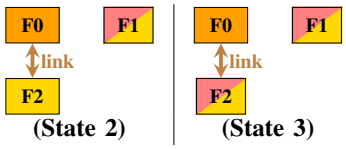
Fig. 2. Linear Temporal Logic Grammar Expansion

means that how AUnit supports the declaration of a valuation needs to be updated. To handle the dynamic changes, we update AUnit test cases to include changes between states of the prime operator (‘’’). To illustrate, consider the following segment from the textual test case in Figure 1 (b):

```
5. Protected = F2
6. Protected' = F1 + F2, Protected'' =  F1 + F2
```

The initial state is outlined in `Protected`, the second state is outlined in `Protected'`, and the final loop state is outlined in `Protected''`. Because the final loop state is required to form a lasso, AUnit test cases outline 3 states, of which only two are unique. The last state, the loop state, will always be equivalent to some earlier state. The counterexample being covered will specify which state to repeat as the loop state.

In addition, we update the Alloy command that executes a test case to explicitly outline how many state transitions are present using the `steps` keyword. The `N steps` keyword tells the Analyzer to search for scenarios with up to `N` state transitions, including the looping transition. The use of `steps` is necessary to ensure that the AUnit test case outlines one specific scenario and not multiple scenarios. In Alloy, the default number of steps considered is 10, which means for any test valuation outlining less than 10 steps, it is possible for the Analyzer to find a satisfying scenario with the same state transitions outlined in the test, but after the outlined state transitions, continue on to other transitions not outlined. For example, for our test case in Figure 1 (b) - (d), if the Alloy execution command on line 12 did not contain the constraint “**but 2 steps**” in its scope, then the following can be a scenario found as a solution when the command is executed:



where state 3 extends the scenario in Figure 1 with an additional state transition. By appending “**but N steps**” to the execution command, we eliminate the possibility of accidentally producing scenarios that further extend the outlined test case and ensure a test execution produce *only* to explicilty outlined scenario.

Besides producing a test suite that will achieve a 100% mutation score, another key benefit to generating tests over any non-equivalent mutant is that μAlloyτ can more directly help the user explore if their model is faulty. To illustrate, the equivalence check for the **UOI** mutant outlined earlier in Section III-B will produce the scenario from Figure 1 (b) - (d), which can then be labeled by the end user as invalid. This test case kills the **UOI** mutant, as the test case will be satisfiable for the original model and unsatisfiable for the mutated model. In addition, this test case also reveals to the user that their model is faulty, as the end user should expect that a protected file can never, at the same time, be in the trash. However, since the test case fails the original model, μAlloyτ will actually reveal to the user that this except invalid behavior is actually allowed by their current model. The user can then use the mutant to help debug and correct the faulty behavior. In this case, the user can apply the mutant itself to correct the faulty model.

D. Mutation Testing

μAlloyτ leaves μAlloy’s mutation testing process intact, as AUnit test case execution, and thus the process for determining if a mutant is killed, does not change due to the additional presence of state-based scenarios.

IV. Implementation

This section outlines important implementation details of μAlloyτ, which is released as a command line tool that runs on top of the Analyzer v.6.0.0.

A. Mutant Operator Subgroups

μAlloyτ uses AUnit’s parser that breaks down an Alloy modeling into its corresponding abstract syntax tree (AST). The built-in Alloy parser contains only enough information to facilitate a translation from Alloy’s logic to an equivalent conjunctive normal form formula, which is needed for the backend SAT solver. However, to perform mutation testing, we need more nuanced information, e.g. it is beneficial for us to know that “`no Protected & Trash`” is a unary *formula* which will evaluate to either true or false while “`Protected & Trash`” is a binary *expression* which will evaluate to a set. While μAlloyτ distinguishes between the type of constraint, expression, or formula, Alloy only distinguished between the operator arity, e.g. “`Protected & Trash`” would be stored as a binary formula since “`& `” is a binary operator.

Each node type in μAlloyτ’s AST additionally stores information about the group of operators for the purpose of creating a balance between the number of mutants generated and the breath of a particular mutant operator. For example, a unary formula can involve any of the following valid unary operators from Alloy’s grammar:

```
unOp ::= ! | not | no | lone | one | some | always |
  eventually | after | before | historically | once
```

which are broken down into the following groups for the purpose of mutation testing:

```
    negation ::= ! | not
multiplicity ::= no | lone | one | some
    temporal ::= always | eventually | after |
                 before | historically | once
```

These groups impact how mutant operators are applied, often by limiting the scope. For instance, for **UOR**, replacements are only done with operators from the same group. This tradeoff ensures we generate mutants that could be realistic mistakes a developer might make, such as using the wrong multiplicity operator, instead of bloating the number of mutants we generate with unlikely mistakes, such as putting a multiplicity operator when the developer should have put a temporal operator.

B. Mutation Visitor and Mutation Operators

To traverse over the AST and apply relevant mutation operators, μAlloyτ uses a Visitor pattern. For each type of node encountered in the AST, μAlloyτ's mutation visitor overrides the visit method and determines which mutation operator rules to invoke by considering the type of node and information about the node's parent in the AST. For instance, if a signature node is encountered, μAlloyτ's mutation visitor applies the **MOR**, **VOI**, and **VOD** mutant operator rules to that location. Besides making different nodes aware of temporal mutants, μAlloyτ significantly changes the **UOI** operator as mentioned in Section III-A. Correspondingly, μAlloyτ changes the conditinos under which this operator is applied to an AST node. In μAlloy, the **UOI** operator only inserted closure ($\hat{}$ and $*$) and transpose (~) operators on any unary expression node encountered. This meant the set multiplicity constraints (`no`, `lone`, `one`, and `some`) were never inserted. This was a design choice made for μAlloy, as inserting the set multiplicity constraints does take a unary expression and transforms it into a unary formula, which frequently causes a compilation error. However, our experiments for μAlloyτ found that broadening the **UOI** operator to have a wider range of insertions was beneficial for detecting more faulty models.

V. Evaluation

We evaluate μAlloyτ over two benchmarks. First, we explore the performance over a benchmark of temporal Alloy models released with the update that added linear temporal logic support [10]. Second, we consider a collection of real-world faulty temporal Alloy models pulled from Alloy4Fun exercises. Alloy4Fun [20] is an online learning platform for Alloy whose exercises have users attempt to write predicates for various models, which are checked against a back-end oracle solution. Submissions to Alloy4Fun have been anonymized and made into an open-source benchmark [5]. For our evaluation, we focus on the two models from Alloy4Fun that include temporal logic (**Trash** and **Train**) that together produce the 3,669 submissions across 37 different exercises: 20 predicates for **Trash** and 17 for **Train**. From there, we remove any duplicate submissions. This results in 2,307 **Trash** models and 683 for **Train** models.

TABLE III
Size of evaluation models.

Model	#Sig	#Rel	#Var	#Cls	#Scp	#M
buffer	4	2	31	1060	3	1
leader	2	4	65	2424	4	1
l_events	3	4	70	3705	4	1
trash	2	0	7	154	3	1
train	7	3	49	880	5	683
trash	3	1	19	194	3	2,307

Table III gives an overview of the size of each model. Column **#Sig** is the number of signatures, column **#Rel** is the number of relations, and column **#Pred** is the number of predicates. Columns **#PVar** and **#Cls** are the number of primary variables and clauses generated for the SAT solver when the empty command is run on the model. Column **Scp** conveys the scope used in the evaluation. Lastly, column **#M** shows how many model variants are considered. For the Alloy 6 models, there is only one version. For the Alloy4Fun benchmark, this is the number of unique faulty submissions.

We address the following research questions:

- **RQ1:** What is the overhead of performing mutation testing of temporal models?
- **RQ2:** How often are mutants of temporal models equivalent mutants?
- **RQ3:** How effective is a mutant-based test suite at detecting faulty temporal models?

All experiments are performed on Linux Ubuntu 20.04 LTS with 1.8GHz Intel i7 CPU and 16 GB RAM.

A. RQ1: Overhead of μAlloyτ

Table IV shows μAlloyτ's performance. Column **Model** reflects the model under evaluation. For the Alloy4Fun models, all faulty submissions for the same predicate are combined together in a single row. For each metric, we present the average across all submissions for the associated exercises, in addition to the minimum and maximum value encountered. The next 4 columns relate to performance metrics for the mutation generation portion of μAlloyτ. Column **#E** shows the number of equivalent mutants, **#NE** shows the number of non-equivalent mutants, and **#Test** shows the number of unique test cases. Lastly, **#T[s]** displays the total generation time in seconds. Since the counterexample that outlines the mutant killing test case is the byproduct of checking whether a mutant is equivalent, we do not separate our test generation time from the overall mutant generation time. The next two columns relate to performance metrics for the mutation testing portion of μAlloyτ. Column **Score** displays the mutation score as a percentage out of 100, and column **T[s]** displays the total generation time in seconds. By design, μAlloyτ's test suite produces a mutation score of 100; therefore, we do not present a range for the **Score** column. As a sanity check, we confirm that all test suites do achieve this score.

Across all models, the time it takes to generate mutants correlates with the total number of mutants explored. This is expected, as for every well-formed mutant that compiles, μAlloyτ calls the SAT solver to check for equivalence, which

TABLE IV
PERFORMANCE RESULTS FOR μALLOYτ

Model	Mutant Generation												Mutation Testing			
	# EQ			#Non-EQ			#Test			Time [s]			Score	Time[s]		
	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max		Avg	Min	Max
buffer	229			202			93			129.75			100	14.20		
leader	360			297			115			618.53			100	40.80		
levents	351			335			170			953.62			100	91.24		
trash	134			133			48			17.95			100	4.89		
trash1	13	1	34	29	16	49	17	13	29	4.38	0.19	13.97	100	0.59	0.31	1.12
trash2	11	1	43	39	16	61	22	13	34	4.77	0.20	22.83	100	0.81	0.30	1.47
trash3	5	2	8	37	25	51	26	19	34	1.32	0.68	1.89	100	0.83	0.51	1.14
trash4	14	2	44	51	25	103	33	18	61	4.19	0.65	31.57	100	1.28	0.49	3.75
trash5	11	1	48	45	16	86	29	13	56	3.05	0.20	57.94	100	1.09	0.30	3.68
trash6	17	1	80	53	16	105	34	13	61	12.58	0.21	696.59	100	1.38	0.30	4.51
trash7	7	2	17	43	25	58	29	19	44	1.94	0.64	4.96	100	0.99	0.50	1.80
trash8	15	1	51	58	16	98	40	13	69	11.20	0.21	961.91	100	1.58	0.30	3.93
trash9	11	2	32	51	23	81	32	16	54	3.75	0.70	14.10	100	1.24	0.46	2.46
trash10	16	1	93	52	16	109	33	13	63	7.37	0.19	234.05	100	1.32	0.31	3.48
trash11	8	2	38	52	25	88	32	17	59	1.93	0.70	5.19	100	1.30	0.50	3.10
trash12	20	1	76	62	16	100	41	13	63	16.83	0.19	451.53	100	1.69	0.31	3.97
trash13	22	1	52	52	16	82	33	13	53	17.46	0.21	120.16	100	1.28	0.33	2.35
trash14	10	1	52	61	16	109	38	13	74	2.75	0.21	23.22	100	1.65	0.33	4.27
trash15	13	1	41	45	16	67	29	13	45	2.72	0.21	6.30	100	1.03	0.32	1.95
trash16	17	5	54	49	23	77	30	16	50	6.84	1.14	24.25	100	1.21	0.43	2.50
trash17	24	1	73	57	16	107	36	13	80	20.12	0.21	483.66	100	1.53	0.33	4.60
trash18	15	1	52	61	16	108	41	13	73	19.40	0.20	629.71	100	1.75	0.31	4.70
trash19	14	5	49	59	36	99	40	22	58	11.87	1.44	105.64	100	1.66	0.69	3.94
trash20	9	1	35	63	16	78	43	13	56	44.33	0.20	1270.18	100	1.92	0.32	2.92
train1	128	117	148	191	167	210	104	91	120	20.99	12.31	50.18	100	14.98	11.60	18.45
train2	127	122	145	191	173	210	106	95	113	16.44	12.36	32.78	100	15.20	12.12	17.38
train3	139	117	208	190	167	245	102	91	134	22.47	11.96	419.64	100	14.60	11.34	22.53
train4	133	117	195	189	167	237	104	91	137	22.80	13.43	194.63	100	14.67	11.71	22.75
train5	146	117	279	235	167	346	133	91	214	21.71	13.40	74.33	100	22.70	11.56	48.12
train6	159	117	252	197	167	255	105	91	143	20.92	12.52	53.25	100	15.70	11.79	25.73
train7	135	123	169	204	173	229	113	93	127	19.29	15.26	48.72	100	17.00	11.96	20.99
train8	148	124	188	239	185	295	137	97	191	148.36	16.73	1065.64	100	24.03	13.77	39.16
train9	143	122	193	209	182	249	118	96	149	52.01	12.83	848.29	100	18.11	13.26	27.23
train10	127	121	133	203	192	215	109	106	113	15.66	14.55	16.77	100	16.15	14.84	17.45
train11	136	130	147	212	202	225	117	111	126	26.80	22.43	30.03	100	18.15	16.81	20.00
train12	117	117	117	167	167	167	91	91	91	13.44	13.44	13.44	100	11.65	11.65	11.65
train13	133	128	144	217	194	232	119	105	130	28.07	16.36	56.62	100	18.90	14.97	21.88
train14	142	126	188	242	204	265	141	114	164	23.45	16.89	42.71	100	24.87	17.75	31.13
train15	138	125	178	203	191	229	114	103	130	21.61	15.78	75.44	100	17.14	14.76	21.76
train16	128	126	129	228	215	244	136	125	144	465.26	56.25	1119.94	100	22.57	19.33	24.93
train17	153	117	227	197	167	223	108	91	132	67.96	13.62	300.92	100	15.71	11.69	20.72
AVG	89.71	56.73	108.43	134.15	93.78	156.54	74.17	53.46	93.89	71.36	8.07	259.54	100	11.69	6.45	12.80

inflates the runtime. However, even with the overhead of checking for equivalent mutants, μAlloyτ's mutant generation runtime is not prohibitive. For the larger Alloy 6 models, μAlloyτ takes, on average, 1.2 minutes to generate mutants, with the two largest models (**leader** anf **levents**) taking over 10 minutes. For the Alloy4Fun models, μAlloyτ's mutant generation time is, on average, 9.94 seconds across all the **trash** variants and 59.25 seconds across all the **train** variants. There are also a few Alloy4Fun submissions, 65 (0.03%), that take over 2 minutes to generate mutants. This spike in runtime occurs on models where the SAT solver struggles to efficiently execute the equivalent mutant checks. However, this issue is not commonly encountered across the benchmark.

In contrast, there is a very minimal overhead for performing mutation testing. Across all models, on average, the mutation testing process accounts for only 26% of the total runtime (mutant generation time combined with mutation testing time). For the Alloy 6 models, the mutation testing runtime averages 11.55 seconds. For the Alloy4Fun models, μAlloyτ's mutation testing time is, on average, 18.94 seconds for the **trash** variants and 37.99 seconds for the **train** variants. Past work concluded that AUnit test cases have negligible execution overhead [27], [33], which is further supported by the observed low mutation testing runtimes, as the mutation testing process is predominately executing AUnit tests.

The small collection of models released with the Alloy 6 update is of a larger scale. Each model comprises of multiple predicates. In contrast, the Alloy4Fun benchmark models all consist of just the faulty predicate along with the high-level model structure. As a result, it is expected that the mutant generation and mutation testing runtimes for the Alloy 6 release models would be larger, as observed. Within the Alloy4Fun benchmark, the *train* model is, on average, more complex than the *trash* model, with notably larger oracle solutions for the predicate exercises. The performance results for μAlloyτ highlight that the overhead of μAlloyτ is related to the size of the model, which makes sense, as a longer model physically means there are more overall mutants to produce. However, the results in Table IV highlight that even for complex models, μAlloyτ's overhead is reasonable.

Altogether, the results demonstrate that mutation testing is relatively inexpensive for temporal models. Given the fact that this overhead includes producing a test suite that will always achieve a 100% mutation score, μAlloyτ is a valuable tool for users to test their models. This is especially true for novice users, given the ability of this test suite to detect faults in models, as seen in Section V-C.

B. RQ2: Equivalent Mutant Rates

μAlloyτ is able to detect how often a valid mutant, a mutant that successfully compiles, is equivalent to the original model or not. For the Alloy 6 models, non-equivalent mutants account for 68.2% of the total number of mutants. Broken down by the origin of the model, non-equivalent mutants account for 47.7% of the Alloy 6 mutants, 79.2% of the trash model mutants, and 60.0% of the train model mutants.

Given the focus on adapting μAlloyτ to temporal logic, we separated out the performance of temporal-based mutants. Any mutant produced by **POD**, **POI**, **VOD**, and **VOI** are automatically considered temporal mutants. In addition, we counted any mutant that changes a temporal operator. For instance, for the **UOR** operator, a mutant that changes "`always x`" to "`until x`" would count as a temporal model. Table V displays information about equivalent versus non-equivalent mutants for temporal mutants under the column **μAlloyτ - τ Only**. Under this column, **#E** is the number of equivalent temporal mutants, **#NE** is the number of non-equivalent temporal mutants, and **%NE** is the percentage of mutants that are non-equivalent out of the total number of temporal mutants generated. In addition, to help facilitate a comparison, the next 4 columns reason over all mutants created by μAlloyτ (**μAlloyτ Full**). **%NE(s)** is μAlloyτ's percentage of static mutants that are non-equivalent, which is any mutant that was not labeled temporal. The last 3 columns depict percentages of temporal mutants to overall mutants for a number of performance metrics. **%τ** is the percentage of total mutants that are temporal, **%NEτ** is the percentage of all non-equivalent mutants that are temporal, and **%τRT** is the percentage of the runtime spent generating temporal mutants.

The results indicate that the rate at which temporal mutants were equivalent versus non-equivalent did not significantly differ compared to the total collection of mutants μAlloyτ produces, although temporal-based mutants are slightly more likely to be equivalent. To illustrate, across all models, for temporal-based mutants, the average percentage of generated mutants that are non-equivalent is 62.6% while the percentage of mutants that focus on changing static operators produces a 71.3% non-equivalent rate on average. At a more detailed level, for the Alloy 6 models, the non-equivalent percentage is 47.1% for temporal-based mutants and 47.8% static mutants. For all trash Alloy4Fun submissions, the non-equivalent percentage is 73.7% for temporal-based mutants and 83.4% tatic mutants. Lastly, for all train Alloy4Fun submissions, the non-equivalent percentage is 53.2% for temporal-based mutants and 62.5% tatic mutants. Across the board, temporal-based mutants account for 33.5% of the total number of mutants and account for 28.0% of the total runtime. Therefore, temporal-based mutants do not seem to have any abnormal impact on the overhead of generating a mutant at large.

TABLE V
EQUIVALENT MUTANT RATES

Model	μAlloyτ - τ Only			μAlloyτ Full			
	#E	#NE	%NE	%NE(s)	%τ	%τNE	%τRT
buffer	70	65	48.1	46.3	31.3	24.3	28.9
leader	99	88	47.1	44.5	28.5	22.9	31.8
levents	89	90	50.3	48.3	26.1	21.2	28.9
trash	36	27	42.9	52.0	23.6	16.9	44.4
trash1	9	13	59.1	80.0	52.4	31.0	36.1
trash2	10	23	69.7	94.1	66.0	37.1	42.6
trash3	3	17	85.0	90.9	47.6	31.5	35.7
trash4	7	18	72.0	82.5	38.5	26.1	30.0
trash5	6	17	73.9	84.8	41.1	27.4	30.5
trash6	9	17	65.4	81.8	37.1	24.3	31.2
trash7	3	16	84.2	87.1	38.0	27.1	31.2
trash8	7	21	75.0	82.2	38.4	26.6	30.1
trash9	5	18	78.3	84.6	37.1	26.1	29.2
trash10	8	18	69.2	81.0	38.2	25.7	32.4
trash11	3	16	84.2	87.8	31.7	23.5	24.4
trash12	11	23	67.6	81.3	41.5	27.1	32.7
trash13	10	18	64.3	73.9	37.8	25.7	27.6
trash14	3	18	85.7	86.0	29.6	22.8	23.8
trash15	7	17	70.8	82.4	41.4	27.4	29.3
trash16	8	17	68.0	78.0	37.9	25.8	25.9
trash17	11	20	64.5	74.0	38.3	26.0	32.6
trsh18	7	21	75.0	83.3	36.8	25.6	29.0
trash19	6	22	78.6	82.2	38.4	27.2	28.2
trash20	5	25	83.3	90.5	41.7	28.4	34.6
train1	37	40	51.9	62.4	24.1	17.3	20.9
train2	38	44	53.7	62.3	25.8	18.7	20.2
train3	44	40	47.6	61.2	25.5	17.4	20.4
train4	40	40	50.0	61.6	24.8	17.5	21.0
train5	42	50	54.3	64.0	24.1	17.5	19.0
train6	49	40	44.9	58.8	25.0	16.9	20.0
train7	43	50	53.8	62.6	27.4	19.7	22.9
train8	45	57	55.9	63.9	26.4	19.3	25.9
train9	49	53	52.0	62.4	29.0	20.2	30.8
train10	38	46	54.8	63.8	25.5	18.5	18.9
train11	43	53	55.2	63.1	27.6	20.0	23.0
train12	32	32	50.0	61.4	22.5	16.1	16.9
train13	44	65	59.6	63.1	31.1	23.0	28.4
train14	44	60	57.7	65.0	27.1	19.9	20.8
train15	46	50	52.1	62.4	28.2	19.8	22.5
train16	41	67	62.0	64.9	30.3	22.7	36.4
train17	56	52	48.1	59.9	30.9	20.9	29.0
AVG	28.4	6.2	62.6	71.3	33.5	23.2	28.0

Overall, we find that the number of equivalent mutants generated is frequently around 40% of the total number of non-equivalent mutants and does not seem to vary too much based on the type of mutant (temporal vs static). In addition, the generation of temporal-based mutants does not notably impact the runtime overhead of mutant generation.

C. RQ3: Effectiveness of Fault Detection

We compare two different settings for μAlloyτ. For configuration "**Grouping**," the default subgroups are applied to the mutation generation process, e.g. **UOR** will consider unary temporal operators separate from unary set operators. For configuration "**Combined**," the subgroups are ignored, and the mutation generation process considers all possible operators, e.g. **UOR** will replace unary temporal and unary set operators interchangeably. The latter configuration is the viewpoint taken by TAR, a mutation-based automated repair technique for Alloy [11]. TAR's mutant operators reason only over predicates and is thus a subset of μAlloyτ's mutant

TABLE VI
COMPARISON OF ABILITY TO DETECT FAULTS IN MODELS.

Model	Grouping						Combined						Diff C - G				
	#Mut	#Test	Time	#D	#T	Pct	#Mut	#Test	Time	#D	#T	Pct	#Mut	#Test	Time	#D	Pct
trash1	29	17	4.38	13	21	61.90	32	17	5.30	13	21	61.90	3	0	0.92	0	0.00
trash2	39	22	4.77	89	104	85.58	46	22	5.74	89	104	85.58	7	0	0.97	0	0.00
trash3	37	26	1.32	3	4	75.00	44	26	1.74	3	4	75.00	7	0	0.42	0	0.00
trash4	51	33	4.19	68	91	74.73	61	34	5.27	68	91	74.73	10	1	1.08	0	0.00
trash5	45	29	3.05	221	274	80.66	54	30	3.86	221	274	80.66	9	1	0.81	0	0.00
trash6	53	34	12.58	207	221	93.67	65	34	14.77	207	221	93.67	12	0	2.19	0	0.00
trash7	43	29	1.94	23	32	71.88	50	30	2.62	23	32	71.88	7	1	0.68	0	0.00
trash8	58	40	11.2	215	219	98.17	74	41	15.17	215	219	98.17	16	1	3.97	0	0.00
trash9	51	32	3.75	67	69	97.10	62	32	4.98	67	69	97.10	11	0	1.23	0	0.00
trash10	52	33	7.37	203	215	94.42	63	33	9.63	203	215	94.42	11	0	2.26	0	0.00
trash11	52	32	1.93	111	113	98.23	69	33	2.99	112	113	99.12	17	1	1.06	**1**	**0.88**
trash12	62	41	16.83	301	311	96.78	75	42	20.23	301	311	96.78	13	1	3.40	0	0
trash13	52	33	17.46	35	38	92.11	64	34	23.84	35	38	92.11	12	1	6.38	0	0
trash14	61	38	2.75	90	91	98.90	78	39	3.99	91	91	100.00	17	1	1.24	**1**	**1.10**
trash15	45	29	2.72	38	41	92.68	54	29	3.49	40	41	97.56	9	0	0.77	**2**	**4.88**
trash16	49	30	6.84	82	86	95.35	61	30	9.08	82	86	95.35	12	0	2.24	0	0
trash17	57	36	20.12	87	93	93.55	72	37	24.62	87	93	93.55	15	1	4.50	0	0
trash18	61	41	19.4	164	165	99.39	75	42	22.99	164	165	99.39	14	1	3.59	0	0
trash19	59	40	11.87	54	56	96.43	71	40	14.31	54	56	96.43	12	0	2.44	0	0
trash20	63	43	44.33	62	63	98.41	75	45	51.86	62	63	98.41	12	2	7.53	0	0
train1	191	104	20.99	35	43	81.40	253	119	32.23	35	43	81.40	62	15	11.24	0	0
train2	191	106	16.44	46	64	71.88	253	120	25.19	46	64	71.88	62	14	8.75	0	0
train3	190	102	22.47	119	150	79.33	256	117	34.24	119	150	79.33	66	15	11.77	0	0
train4	189	104	22.8	46	61	75.41	254	118	33.78	46	61	75.41	65	14	10.98	0	0
train5	235	133	21.71	121	129	93.80	326	152	37.00	121	129	93.80	91	19	15.29	0	0
train6	197	105	20.92	62	80	77.50	265	120	33.30	62	80	77.50	68	15	12.38	0	0
train7	204	113	19.29	33	33	100.00	274	128	30.16	33	33	100.00	70	15	10.87	0	0
train8	239	137	148.36	29	31	93.55	332	160	287.34	29	31	93.55	93	23	138.98	0	0
train9	209	118	52.01	81	82	98.78	278	135	77.57	81	82	98.78	69	17	25.56	0	0
train10	203	109	15.66	2	2	100.00	272	123	27.02	2	2	100.00	69	14	11.36	0	0
train11	212	117	26.8	13	16	81.25	282	135	47.86	16	16	100.00	70	18	21.06	**3**	**18.75**
train12	167	91	13.44	0	2	0.00	223	104	20.72	0	2	0.00	56	13	7.28	0	0
train13	217	119	28.07	10	10	100.00	289	135	40.29	10	10	100.00	72	16	12.22	0	0
train14	242	141	23.45	26	26	100.00	334	161	39.56	26	26	100.00	92	20	16.11	0	0
train15	203	114	21.61	31	31	100.00	281	129	33.84	31	31	100.00	78	15	12.23	0	0
train16	228	136	465.26	6	6	100.00	305	157	627.78	6	6	100.00	77	21	162.52	0	0
train17	197	108	67.96	12	15	80.00	266	125	91.43	12	15	80.00	69	17	23.47	0	0
Avg	122.51	67.30	40.70			87.24	161.84	78.67	47.72			87.93	39.32	8.00	15.13		0.69

operators. The "**Combined**" configuration is the same as comparing TAR's ability to detect faults if you only use TAR's first-order mutants and if the following mutation operators are added to TAR (**MOR**, **VOI**, **VOD**, **PBD**, **IEPE**).

Table VI displays the fault detection capability of the two configurations. Column **Model** displays the exercise under consideration: all faulty submissions for the same predicate are combined together in a single row, and the average number for each metric is reported. The next 6 columns display performance information about the "**Grouping**" configuration. **#Mut** is the number of non-equivalent mutants, **#Test** is the number of tests generated and **Time** is the mutant generation runtime in seconds. This information is repeated from Table IV for easier comparison. Then, column **#D** is the number of faulty submissions detected, **#T** is the total number of faulty submissions, and **Pct** displays the percentage of faulty submissions that were detected. The next 6 columns repeat this information for the combined configuration. Lastly, the next 5 columns present the difference between the configurations. For each column, the value is created by subtracting the combined configuration value from the grouping value.

Across all exercises and both configurations, μAlloyτ is able to detect all the faults for six of the train exercises. For the "**Grouping**" configuration, μAlloyτ is able to detect 90.8% of the faulty models. Outside of **train12**, in which neither configuration detects either of the 2 faulty submissions, the "**Grouping**" configuration detects 62% or more of the fault submissions per exercise and averages a detection rate of 89.75% for the trash exercises, 84.3% for the train exercises for a combined average of 87.24%. To achieve this detection rate, under the "**Grouping**" configuration, μAlloyτ generates on average 67.3 tests: 26.65 tests on average for the trash exercises and 115.12 tests on average for the train exercises. These test suites are automatically created on average in 40.70 seconds: 12 seconds on average for the trash exercises and 74.42 seconds on average for the train exercises.

In addition, although non-equivalent mutants directly influence the tests, we do not find that the rate at which non-equivalent mutants occur impacts the fault detection capability. For example, for **trash7**, μAlloyτ detects 71.8% of the faulty submissions, but 86% of the mutants generated were non-equivalent. For **train15**, μAlloyτ detects 100% of the faulty submissions, but 52.1% of the mutants generated were non-equivalent. Then, for **trash9**, μAlloyτ is only able to detect 97.1% of the faulty submissions, and 78.2% of the mutants generated were non-equivalent. Therefore, there does not appear to be a strong relationship between non-equivalent mutant rates and the likelihood that μAlloyτ will detect a fault.

There is a minor difference when comparing the fault detection capability of the two configurations. The "**Combined**" configuration is about to detect 7 faulty exercises that the grouping configuration failed to, which is an increase in the detection rate of only 0.13% to a total of 90.97% of total faulty submissions detected. This slight increase in fault detection does come at the expense of the overhead of μAlloyτ. On average, in the "**Combined**" configuration, μAlloyτ will generate 1.3× more mutants, which results in 1.1× more test cases getting generated. These test cases do a human to provide the oracle; therefore, the "**Combined**" configuration slightly increases the human effort needed. However, the main detriment is that the runtime increases by 1.42×.

Overall, the results support that μAlloy$\tau$is effective at detecting faults in real-world models. *Moreover, taking all three research questions into account, we believe μAlloyτ can be invaluable to novice users, helping them developer more reliable software models and easing the adoption of Alloy.*

D. Threats to Validity

The Alloy4Fun benchmark is representative of mistakes that novice Alloy users would make; thus, our results for fault detection may not generalize to faulty models made by expert users or non-educationally driven models. While μAlloyτ is intended to benefit all Alloy users, we believe that novice users are more likely to seek out the use of μAlloyτ to build confidence in the accuracy of their model. In addition, while **Trash** can be viewed a toy model, **Train** is a noticeably more robust model that is derived from real-world train systems in Alloy [12]. Therefore, this is a good variety of complexity of predicate formulas throughout the Alloy4Fun benchmark.

VI. Related Work

Testing and Debugging Alloy Models. The most closely related work to μAlloyτ is TAR, a mutation-oriented repair technique that is aimed at repairing Ally4Fun models. There are three main. First, because of the execution environment for Alloy4Fun, TAR does not define any mutation operators over signatures or relations. As a result, TAR can simply use the biconditional operator when checking equivalence. Second, TAR views mutant operators at a higher granularity than μAlloyτ. For instance, for a binary formula, the binary logical operators are combined together into one mutant operator group. Since TAR only mutates faulty locations and does not generate all possible first-order mutants, TAR can afford to make this tradeoff in granularity. Lastly, TAR does not perform mutation testing but instead applies 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, μAlloyτ generates all possible first-order mutants, performs traditional mutation testing, and importantly, produces a test suite that will achieve a 100% mutant score. Since TAR's mutants are a subet of μAlloyτ's mutants, we did not do a direct comparison of mutant operators. However, our combined configuration takes TAR's perspective for grouping mutant operators.

Besides μAlloyτ and TAR, there are several bodies of work that focus on testing and debugging Alloy models. Three make use of AUnit. 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 [32]. ARepair is a generate and validate automated repair technique that uses AUnit test cases to outline expected behavior [30]. ICEBAR extends ARepair to additionally consider built-in Alloy assertions to guide the repair and check candidate patches [15]. There are also several debugging techniques for Alloy that do not utilize AUnit tests. 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. BeAFix is an automated repair technique that uses a bounded exhaustive search [9]. FLACK is a fault localization technique that locates faults by using a partial max sat toolset to compare the difference between a satisfying instance of a predicate and a counterexample from an assertion over that predicate [36]. These techniques focus heavily on using assertions, but assertions have to be written in first-order logic and can be incorrectly specified. None of these frameworks support temporal logic.

Mutation Testing. Mutation testing [13], [16] is an active research area [18] 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], [24], which reason over linear temporal logic but the automated analysis is different. Our work focuses on bounded scenario enumeration while model checkers perform an exhaustive search of the state space, which results in the execution and checking of mutations requiring different algorithmic approaches. Often, mutation testing for declarative languages focuses on mutating specifications for imperative code, often with mutations applicable to both the specification and the imperative code. Our work on mutation testing for Alloy is closest in spirit to Srivatanakil et al. [26] 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. In addition, MuCheck introduces mutation testing for Haskel [19].

VII. Conclusion

Alloy 6 introduces linear temporal logic to Alloy, making Alloy a more versatile modeling language. However, this increased expressibility comes at a cost: there are more ways users can incorrectly specify a property. μAlloy is a mutation testing framework for Alloy 5 and earlier. μAlloy both performs mutation testing and has the capability of generating a mutant-killing test suite. This paper introduces μAlloyτ, which expands μAlloy to perform mutation testing over temporal models. μAlloyτ also updates AUnit to define what a temporal test case looks like. Experimental results show that μAlloyτ has a minor overhead and the test suites produced by μAlloyτ are effective at revealing several faults in real-world faulty temporal models. Moreover, μAlloyτ opens to door to update AlloyFL and ARepair to handle the new temporal constructs.

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] Brida, S.G., Regis, G., Zheng, G., Bagheri, H., Nguyen, T., Aguirre, N., Frias, M.F.: Bounded exhaustive search of alloy specification repairs. In: ICSE (2021)

[10] Brunel, J., Chemouil, D., Cunha, A., Macedo, N.: The electrum analyzer: Model checking relational first-order temporal specifications. In: 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 884–887 (2018). https://doi.org/10.1145/3238147.3240475

[11] Cerqueira, J., Cunha, A., Macedo, N.: Timely specification repair for alloy 6. In: Software Engineering and Formal Methods. pp. 288–303 (2022)

[12] Cunha, A., Macedo, N.: Validating the hybrid ertms/etcs level 3 concept with electrum. Int. J. Softw. Tools Technol. Transf. **22**(3), 281–296 (jun 2020)

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

[14] 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)

[15] 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)

[16] 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

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

[18] 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

[19] Le, D., Alipour, M.A., Gopinath, R., Groce, A.: Mucheck: An extensible tool for mutation testing of haskell programs. In: Proceedings of the 2014 international symposium on software testing and analysis. pp. 429–432 (2014)

[20] Macedo, N., Cunha, A., Pereira, J., Carvalho, R., Silva, R., Paiva, A.C.R., Ramalho, M.S., Silva, D.: Experiences on teaching alloy with an automated assessment platform. In: Raschke, A., Méry, D., Houdek, F. (eds.) Rigorous State-Based Methods. pp. 61–77 (2020)

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

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

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

[24] 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)

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

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

[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, K., Sullivan, A., Marinov, D., Khurshid, S.: ASketch: a sketching framework for Alloy. In: 6th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z (ABZ). pp. 121–136 (2018). https://doi.org/10.1007/978-3-319-91271-4_9

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

[35] Zheng, G., Nguyen, T., Brida, S.G., Regis, G., Aguirre, N., Frias, M.F., Bagheri, H.: Atr: Template-based repair for alloy specifications. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. p. 666–677. ISSTA 2022 (2022)

[36] 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)