Timely specification repair for Alloy 6*

Jorge Cerqueira[1,2][0000−0002−7570−872X], Alcino Cunha[1,2][0000−0002−2714−8027], and Nuno Macedo[1,3][0000−0002−4817−948X]

[1] INESC TEC, Porto, Portugal
[2] University of Minho, Braga, Portugal
[3] Faculty of Engineering of the University of Porto, Porto, Portugal

Abstract. This paper proposes the first mutation-based technique for the repair of Alloy 6 first-order temporal logic specifications. This technique was developed with the educational context in mind, in particular, to repair submissions for specification challenges, as allowed, for example, in the Alloy4Fun web-platform. Given an oracle and an incorrect submission, the proposed technique searches for syntactic mutations that lead to a correct specification, using previous counterexamples to quickly prune the search space, thus enabling timely feedback to students. Evaluation shows that, not only is the technique feasible for repairing temporal logic specifications, but also outperforms existing techniques for non-temporal Alloy specifications in the context of educational challenges.

Keywords: Specification repair · First-order temporal logic · Formal methods education · Alloy.

1 Introduction

Besides their role in traditional formal methods, namely model checking, formal specifications are becoming central in many software engineering techniques, such as property-based testing, automated program synthesis or runtime monitoring. Therefore, software engineers with little expertise on formal methods are increasingly being required to write and validate formal specifications. Unfortunately, students and professionals still struggle with this task, and more advanced tool support is needed if formal specifications are to be embraced by a wider community [10].

Alloy [9] is a formal specification language supported by automated model finding and model checking, being the quintessential example of a lightweight formal method. Its most recent version 6 [11] is based on a first-order relational temporal logic, enabling both structural and behavioural modeling and analysis. For these reasons, Alloy is often used in formal methods introductory courses[4].

* This work is financed by National Funds through the Portuguese funding agency, FCT – Fundação para a Ciência e a Tecnologia within project EXPL/CCI-COM/1637/2021.

[4] `https://alloytools.org/citations/courses.html`

Alloy4Fun [12][5] is a web-platform for Alloy that supports automated assessment exercises through the creation of specification challenges: instructors write a secret predicate that acts as an oracle, and the students have to write an equivalent predicate given an informal description. If the submitted predicate is incorrect, the student can navigate through counterexamples that witness the inconsistency with the oracle. Unfortunately, in our experience, novice practitioners struggle with interpreting such counterexamples and tracing the problem back to the specification.

The automatic generation of hints to guide students in fixing their code has long been employed in educational coding platforms. One approach to the generation of such hints is to apply automated repair techniques and then derive a hint back from the found sequence of repairs [13]. Although automated repair for specifications is still largely unexplored, recently, a few approaches have been proposed for the previous (non-temporal) version of Alloy, namely ARepair [18] and BeAFix [2]. However, the educational scenario has some characteristics that prevent their adoption for hint generation. The main issue is that their performance (likewise most techniques for code [17]) is still not good enough to support hint generation. Timely feedback is particularly important in this context, to avoid the student hitting bottlenecks and frustration when interacting with the platform. Additionally, ARepair uses test cases as oracles, and it is difficult to manually write a set of test cases that brings its accuracy up to an acceptable level. BeAFix is better suited to repair specification challenges, since it uses the Alloy checks as oracles, but prunes the search space by exploiting multiple suspicious locations and multiple failing oracles, techniques that are useless in this context, where we just want to fix one (usually short) predicate written by the student that failed one specific check against the oracle.

This paper presents a new mutation-based technique for the repair of Alloy 6 specifications that can be used in the educational context for timely hint generation. It is the first repair technique to consider the full logic of Alloy 6, including both its first-order and temporal constructs. Also, it implements a pruning technique based on evaluating previously seen counterexamples, that can be used to optimize repairs in models with a single faulty location, as is the case of specification challenges. Our evaluation shows that the proposed technique considerably outperforms existing automated repair techniques for Alloy (when considering only the first-order subset of the language they support).

The rest of this paper is organized as follows. Section 2 presents existing work on automated specification repair. Section 3 presents the novel specification repair technique and associated pruning strategy, whose performance is evaluated against the existing approaches in Section 4. Lastly, Section 5 draws conclusions and points directions for future work.

[5] `http://alloy4fun.inesctec.pt/`

2 Related Work

There is extensive work on automated program repair [7,6], with techniques being broadly classified as search-based (or generate-and-validate) – which search for possible solutions and test them against the oracle – or semantics-driven (or constraint-based) – where the needed repair is encoded as a constraint from which a correct fix is generated. Most approaches use test cases as oracles, although a few rely on reference implementations (e.g., in the educational context) or program specifications (e.g., in the context of design by contract). In contrast, there is very little work on automated specification repair. In [4] a search-based technique is proposed to fix OCL integrity constraints against existing model instances. SpeciFix [14] is a search-based technique for fixing pre- and post-conditions of Eiffel routines against a set of test cases. Techniques [15,3] for semantics-driven repair in the B-method focus on repairing the state machine rather than the broken specifications. Two techniques – ARepair and BeAFix – have been proposed for automatic repair of Alloy specifications, which we further detail next.

ARepair [18,19] uses test cases as oracle. The downsides of this approach are twofold: it is prone to overfitting, where an accepted fix passes all the tests but not the expected properties; and the user is required write (high quality) unit tests, something that is not common practice for Alloy or specifications in general. ARepair starts by feeding the model and tests into AlloyFL [20], a mutation-based fault localization framework for Alloy, which returns a ranked list of suspicious Abstract Syntax Tree (AST) nodes. Then, it checks if the mutation provided by AlloyFL on the most suspicious node retains currently passing tests and passes some previously failing tests. Otherwise, it creates holes and tries to synthesize code for these holes that make some of the failing tests pass. These tests are performed with Alloy's evaluator, avoiding calls to the solver. This process is repeated until all tests pass. The synthesizer returns complex non-equivalent expressions for a specified type and bounds. Since a huge amount of expressions is synthesized, ARepair presents two search strategies, one which chooses a maximum amount of tries per hole and tries to prioritize certain expressions; and another which iteratively fixes all holes except one for which it it tries all expressions to find the one that makes most tests pass.

In contrast, BeAFix [2,1] uses the check commands of an Alloy specification as oracles, focusing on the repair of the system specification referred to by the check. This is a more natural scenario since defining checks to verify the intended properties of a design are common practice. BeAFix relies on a different fault localization technique for Alloy, FLACK [22], which it only runs once for the initial model, unlike ARepair. To generate the fix candidates, BeAFix defines a set of mutation operators that are then combined up to a certain maximum amount of mutators. Mutated expressions are then tested against the oracles using Alloy's solver. Since the number of candidates grows exponentially with the maximum amount of combined mutations, BeAFix relies on two pruning strategies to discard groups of candidates that are guaranteed to not fix the specification, without calling the solver for a full check. Partial repair checking is

```
var sig File {
    var link : lone File
}
var sig Trash in File {}
var sig Protected in File {}

//SECRET
pred prop4o {
    eventually some Trash
}
//SECRET
check {
    prop4 <=> prop4o
}
// some file will eventually be sent to the trash
pred prop4 {

}
```

Fig. 1. An example specification challenge in Alloy4Fun

used when there is a command $Check_i$ that refers to a suspicious location l_0, but not another suspicious location l_1. If $Check_i$ is still invalid under mutation m_0 for l_0, it is not worth to pair mutations for l_1 with m_0 since they will never render $Check_i$ valid. Variabilization is used when a $Check_i$ fails for a pair of mutations m_0 and m_1 for suspicious locations l_0 and l_1, having yielded a counterexample. To check whether m_0 is a mutation worth exploring, variabilization freezes m_0 and replaces l_1 by an existentially quantified variable and checks whether the counterexample persists for $Check_i$. If so, there is no possible value for l_1 that fixes the specification for m_0 at l_0 and that mutation can be automatically discarded.

3 Alloy Temporal Repair

This sections presents the main contribution of this paper: an automatic repair technique for Alloy 6 temporal specifications, suitable for the educational domain. Required Alloy concepts are introduced as needed, but for a more thorough the reader should consult [8].

3.1 Overview

Our goal is to use automatic specification repair to generate hints to students in autonomous assessment platforms. For Alloy, Alloy4Fun is currently the only framework providing such functionality, by allowing the definition of secret predicates and check commands. A typical usage of this feature is in the creation of

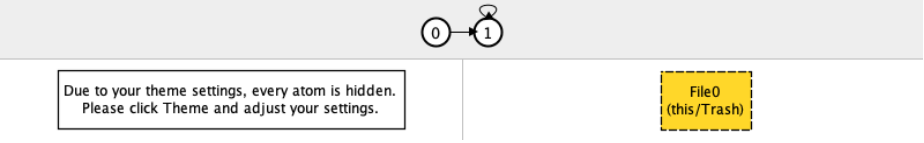

Fig. 2. Counterexample to `prop4`

specification challenges: the instructor writes a hidden predicate representing a correct answer, and a hidden check command that tests it against an initially empty predicate that the student is expected to fill. As an example, consider the Alloy snippet presented in Fig. 1, modelling a simple file system where a file can link to another file, be put in the trash, or be set in a protected state. This snippet belongs to an exercise given to students in a formal methods course at University of Minho, Portugal [12]. The keyword **`sig`** declares a new signature in a specification, grouping together a set of atoms of the domain. Signatures can be defined as subsets of other signatures using keyword **`in`**. Inside signatures, fields can be declared to introduce new relations between atoms, for example the `link` binary relation. Signatures and fields can have multiplicty constraints, such as the one in `link` stating that each file links to at most one other file (**`lone`**). The **`var`** keyword indicates that the content of a signature or field can change between time instances.

Each exercises has multiple specification challenges. The fourth one of this exercise asks the student to write a formula that evaluates to true iff a file is sent to the trash at any point in time. The student is asked to write such a formula in predicate (**`pred`**) `prop4`. Hidden to the student, marked with the special comment `//SECRET`, a check command tests whether the student's predicate is equivalent to the instructor's oracle written in `prop4o`, written using the temporal operator **`eventually`** and cardinality test **`some`**. The most common erroneous solution submitted by the students is the following.

```
pred prop4 {
    some f : File | eventually f in Trash
}
```

Without temporal operators, expressions are evaluated in the initial state, so the outermost existential quantifier is evaluated in the first state. So this predicate is actually stating that a file present in the first state is eventually sent to the trash, disregarding scenarios where a file created after the first state is sent to the trash. Checking against the oracle, Alloy would return a counterexample trace such as the one in Fig. 2, where a file is directly created in the trash in the second state[6]. Students would then interpret the counterexample, trying to find the error in their reasoning.

[6] Note that in this challenge the evolution of the system is not restricted and files are not required to be created before being sent to the trash. The goal of the exercise was precisely to train students to write the weakest specifications of the stated requirement, independent of concrete system implementations.

Search-based automatic repair approaches usually implement a generate-and-validate process: alternative candidate solutions are generated and then tested against the oracle for correctness. Mutation-based approaches generate candidates by mutating nodes of the AST. In this scenario, it is natural to use a repair technique whose oracles are themselves specifications provided in the check commands, rather than test cases, as does BeAFix. For instance, for the incorrect submission above, with search depth 1, mutants like the ones below would be generated and tested against the oracle:

1. `some f : Trash | eventually f in Trash`
2. `some f : File | eventually f in File`
3. `some f : File | always f in Trash`
4. `some f : File | after f in Trash`
5. `some f : File | eventually f not in Trash`
6. `all f : File | eventually f in Trash`
7. `after some f : File | eventually f in Trash`
8. `eventually some f : File | eventually f in Trash`

Of these mutants, only the last one is valid and equivalent to `prop4o`. Note that this approach tests the semantic equivalence of the submissions against the oracle, rather than its AST. This also means that validating a mutant amounts to calling the *solver* to run the check command. Calls to the solver are expensive, and since the number of mutants may be overwhelming, this process in general is infeasible without pruning strategies. Unfortunately, BeAFix's pruning strategies are not effective in this scenario: partial repair can optimize the procedure when there are multiple failing checks, but in this scenario each challenge amounts to a single check; variabilization optimizes the procedure when there are multiple suspicions locations, but here we already know that the suspicious location is the single predicate filled by the student.

The idea behind the pruning strategy proposed in this work is that a counterexample for a candidate mutant will likely be a counterexample for similar candidates. For instance, the counterexample returned for the initial student submission in Fig. 2 would actually discard the invalid mutants 1–5 presented above, avoiding 5 additional calls to the solver. Calling the solver for mutant 6 could return a counterexample with some files in the trash in the first state that are then removed in the second, serving also to discard mutant 7. The principal advantage of this reasoning is that testing a mutant specification over a concrete counterexample does not require calling the solver: it can be performed efficiently with Alloy's *evaluator*. Therefore, by storing counterexamples obtained for previously discarded mutants, new candidates can be quickly checked against them before calling the solver to run the check command.

3.2 Mutation-based repair with counterexample-based pruning

The technique proposed in this work has in common with BeAFix the fact that it generates fix candidates through a set of mutations. The main differences are

Name	Mutation	Example
RemoveBinary	`A [bop] B` ⇝ `A` `A [bop] B` ⇝ `B`	`A and B` ⇝ `A`
ReplaceBinary	`A [bop] B` ⇝ `A [bop'] B`	`A and B` ⇝ `A or B`
RemoveUnary	`[uop] A` ⇝ `A`	`always no A` ⇝ `no A`
ReplaceUnary	`[uop] A` ⇝ `[uop'] A`	`no A` ⇝ `some A`
InsertUnary	`A` ⇝ `[uop] A`	`no A` ⇝ `always no A`
BinaryToUnary	`A [bop] B` ⇝ `[uop] (A [bop'] B)`	`A in B` ⇝ `no (A + B)`
QuantifierToUnary	`[qtop] a:A | B` ⇝ `[uop] A`	`no a:A | foo[a]` ⇝ `no A`
ReplaceQuantifier	`[qtop] a:A | B` ⇝ `[qtop'] a:A | B`	`no a:A | foo[a]` ⇝ `some a:A | foo[a]`

Table 1. List of mutators for Boolean formulas.

twofold: the development of a new pruning technique suitable for specifications with a single check command and suspicious location, and the support for the temporal logic of Alloy 6 that has not been addressed thus far.

In Alloy, a check command with formula ϕ over a specification defined by formula ψ (in Alloy, defined through **fact** constraints) is converted into a model finding problem for a single formula ϕ **and not** ψ. For instance, in the challenge from Fig. 1 ψ is empty, so for the `prop4` example shown in Section 3.1 the check's formula would simply be converted to

```
not ((some f : File | eventually f in Trash) <=>
     (eventually some Trash))
```

When such a formula is passed to the solver, if the check is invalid it will return a counterexample c (such as the one in Fig. 2 for the example) where the specification facts ψ hold but the check ϕ does not. If there is no such counterexample, the check holds and the solver returns $\perp$. Alloy 6's analyzer checks assertions either with SAT solvers or SMV model checkers, the former only for bounded model checking. Although Alloy's logic is first-order, such analysis is possible because there is a bound imposed on the size of the universe by defining *scopes* for signatures (the default scope is 3). For bounded model checking, the default analysis for temporal properties, it is also possible to define a scope for the temporal horizon (the default being 10 steps). For a concrete counterexample c, Alloy also provides an evaluator that can efficiently calculate the value of any formula ϕ without calling the solver, which simply returns true or false.

A mutation m of a formula ϕ is simply a pair (l, o) of a location l in ϕ (which can be seen as a path through the AST, identifying a concrete node) and an instantiation of a mutator o from Tables 1 or 2 (to be presented shortly). These mutations m are uniquely identified by the location and operation. Each candidate mutant results from the application of a sequence δ of such mutations to the specification that is to be fixed. Order within δ is relevant since a mutation m_1 may refer to a location introduced by a preceding mutation m_0, and a mutation m_1 cannot refer to a location previously removed by a preceding m_0.

Name	Mutation	Example
RemoveBinary	`A [bop] B` ⇝ `A` `A [bop] B` ⇝ `B`	`A + B` ⇝ `A`
ReplaceBinary	`A [bop] B` ⇝ `A [bop'] B`	`A + B` ⇝ `A - B`
RemoveUnary	`[uop] A` ⇝ `A`	`~A` ⇝ `A`
ReplaceUnary	`[uop] A` ⇝ `[uop'] A`	`^A` ⇝ `*A`
InsertPrime	`A` ⇝ `A'`	`A` ⇝ `A'`
InsertBinary	`A` ⇝ `A [bop] B`	`A` ⇝ `A + B`
InsertUnary	`A` ⇝ `[uop] A`	`A` ⇝ `~A`
ReplaceRelation	`A` ⇝ `B`	`A` ⇝ `B`

Table 2. List of mutators for relational expressions.

A procedure that we abstract as MUTATE takes a specification ϕ and a location l and generates all possible mutations for all AST nodes below l. In our example, l would identify the sub-formula that resulted from the student's submitted predicate (the left-hand side of the equivalence). This procedure is lazy, returning an iterator Δ to generate new mutations on demand. Procedure APPLY represents the actual application of a sequence of mutations δ to a specification, returning a new specification mutant. The skeleton of the available mutation operations are presented in Tables 1 and 2 for Boolean formulas – composed of Boolean connectives, first-order quantifications and temporal operators – and relational expressions – composed of relational operations, transitive closure and temporal primes –, respectively. Mutators are guaranteed to not change the type of an expression, so, for instance, RemoveUnary for Boolean formulas cannot remove a multiplicity test operator, since its sub-expression is a relational expression, and an operator is always replaced by another of the same type (i.e., Boolean connectives cannot be replaced by relational operators). Operations that require the insertion of relational expressions (namely InsertBinary and ReplaceRelation) only introduce a single relation at a time and takes into account type information to avoid creating expressions considered irrelevant according to Alloy's type system [5]. For instance, InsertBinary for an expression `A` only creates intersection expressions `A & B` for relations `B` whose type has some elements in common with the type of `A`. For the particular case of introducing a join operator, InsertBinary also only explores mutations that preserve the arity of the original relational expression.

An abstract view of the repair procedure is shown in Algorithm 1. The procedure registers a set *cands* of candidate sequences of mutations δ. At each depth level, the procedure iterates over all *cands* and adds an additional mutation m to a candidate δ. Procedure MUTATE is called over ϕ already mutated with the previous candidate, so that only mutations over valid locations are generated (in case locations from the original ϕ_0 were removed by δ, or new ones introduced). Moreover, to avoid testing redundant mutants, whenever a new candidate is generated it is only analyzed if it has not been previously seen in *cands*. Although abstracted in Algorithm 1, this membership test $\in$ ignores the order of the mu-

```
Input: A formula φ0 representing an invalid check and a location l in φ0 to fix.
Output: A passing formula or ⊥
cands ← {[]};
cexs ← [(SOLVE(φ0), 1)];
for d ∈ 1 ... MAXDEPTH do
    cands′ ← ∅;
    while cands ≠ ∅ do
        δ ← cands.POP();
        Δ ← MUTATE(APPLY(φ0, δ), l);
        while Δ.HASNEXT() do
            δ′ ← δ ++[Δ.NEXT()];
            if δ′ ∉ cands then
                φ ← APPLY(φ0, δ′);
                valid ← TRUE;
                cexs′ ← cexs.CLONE();
                while cexs′ ≠ ∅ ∧ valid do
                    c ← cexs′.PULLHIGHEST();
                    valid ← EVALUATE(φ, c);
                    if ¬valid then cexs.INCPRIORITY(c);
                if valid then
                    c ← SOLVE(φ);
                    if c = ⊥ then return φ;
                    else cexs.PUSHPRIORITY(c, 1);
                cands′.PUSH(δ′);
    cands ← cands′;
return ⊥;
```

Algorithm 1: Repair procedure with counterexample-based pruning

tations, meaning that two sequences $[m_0, m_1]$ and $[m_1, m_0]$ are considered the same. This is sound because we assume that a candidate δ cannot contain more than one mutation for the same location l.

Without counterexample-based pruning, for each candidate δ' the procedure would simply calculate a mutant ϕ as APPLY(ϕ_0, δ') and call the solver (here, procedure SOLVE(ϕ)). The procedure stops when a specification ϕ is valid according to the SOLVE, or the maximum search depth MAXDEPTH is reached, returning $\perp$. In the example previously presented, the INSERTUNARY mutator can be applied to obtain the expression `eventually some f : File | eventually f in Trash`. The next most common incorrect submission is

```
some f : File | eventually always f in Trash
```

which, besides the same problem of only quantifying on the files available in the first state, also assumes that a file in the trash must stay there indefinitely (temporal operator **always**). This requires search level 2: one mutation to add an outer-most **eventually**, and another to remove the **always** through REMOVEUNARY. The third most common is

```
eventually File in Trash
```

which incorrectly states that at some point in time, all existing files are in the trash. It can be fixed through a single application of BinaryToUnary, resulting in `eventually some File & Trash`, which is yet another formula equivalent to `prop4o`.

To avoid expensive calls to the solver, our technique's pruning strategy first evaluates the candidate formula against previously seen counterexamples. These are kept in a priority queue *cexs*, where the priority of each counterexample c is the amount of candidates they were able to prune. So for each mutant, the evaluator is called (procedure EVALUATE) to test ϕ for every previously found counterexample following the established priority. If ϕ still holds for a counterexample c, then it is still an invalid mutant, so ϕ is discarded and c has its priority increased in *cexs*. Only after passing all previously seen counterexamples is the solver called for ϕ. If the solver returns $\perp$, then ϕ has effectively been fixed. Otherwise, a new counterexample c is found and added to *cexs* with minimal priority.

3.3 Implementation details

To improve performance, richer data structures were used in the implementation of Algorithm 1. To avoid repeating the generation of all mutations for all candidate mutants that have ASTs that are still very similar to each other, MUTATE is not freshly called for every candidate. Instead the candidates are stored in a list with a pointer to their predecessor candidate. Thus, to generate all the candidates up to a depth, the index to the candidate being checked is kept, as well as the candidate that generated the latest added candidates. When the end of generated candidates is reached, more are generated from the candidate after the one the latest candidates were generated from. The last counterexample used to prune is also tracked and is tested first, the reasoning being that candidates coming after one another will likely mutate the same locations, and thus, also be more likely to be pruned by the same counterexamples. To prevent combining mutators that would generate incorrect or repeated candidates (for example, when sub-expressions are removed), a mutation also registers blacklisted locations that can no longer be mutated. Lastly, rather than just keep track of mutations δ in *cands*, we also maintain the associated mutant ϕ to avoid re-applying mutations.

The technique was implemented as an extension of Alloy 6. It does not make modifications to original Alloy 6 source code. Instead, it only adds new packages and uses the public methods of the original, hopefully making it easier for anyone to follow the implementation and to update to future Alloy releases. The source code is public and can be found on GitHub[7], as well as a Docker container[8] to allow easier replication of the results. In the implementation, the user has to specify by hand the suspicious predicate that is to be fixed. However,

[7] `https://github.com/Kaixi26/TAR`
[8] `https://hub.docker.com/r/kaixi26/tar`

the technique itself has no limitations in terms of compatibility with fault localization techniques, and could have been paired with one of those techniques to automatically identify such predicate.

4 Evaluation

In this section we evaluate the performance of the proposed technique for timely Alloy 6 repair (TAR, in the presented results), with the goal of answering the following research questions:

RQ1 What is the performance of mutation-based repair with counterexample-based pruning for temporal Alloy 6 specifications?
RQ2 How does its performance compare with that of existing automatic repair techniques for static Alloy specifications?
RQ3 What is the actual impact of counterexample-based pruning?

Alloy4Fun stores all submissions made to challenges. These are available to the creators of the challenges for subsequent analysis. Tutors at the Universities of Minho and Porto have been using Alloy4Fun in classes for several years and publicly share the data after anonymization[9]. These challenges follow the shape of the one in presented in Fig. 1, so it is easy to identify the oracle and the student predicate to be repaired in each submission. Thus, for RQ1 we executed TAR for all erroneous submissions to challenges with mutable relations (only allowed in Alloy 6) in the 2021 Alloy4Fun dataset. This amounts to two exercises (`TrashLTL` and `Trains`) composed of 38 challenges, totalling 3671 submissions. These results are summarized in Fig. 3, for different search depth levels, and also in the bottom part of Table 3. BeAFix also used a subset of Alloy4Fun challenges for their evaluation [2] (those compatible with the previous Alloy 5 version). For RQ2, we've also run TAR for submissions to those purely first-order logic challenges. This amounts to 6 exercises (`TrashRL`, `ClassroomRL`, `CV`, `Graphs`, `LTS` and `Production`) composed of 48 challenges with 1935 submissions. ARepair requires the user to specify test cases which are not available for the Alloy4Fun challenges, but writing them ourselves could introduce a bias in the process. Instead, we used counterexamples generated during the counterexample-based pruning process as test cases. For each student submission for a challenge, counterexample-based pruning iterated over a set of counterexamples until a fix was found. Counterexamples more commonly occurring in this process have contributed to fixing the most incorrect submissions, and thus are representative of the challenge. We ran ARepair for the same structural Alloy4Fun challengs as BeAFix using the top 10 and top 25 counterexamples as test cases. For the comparison of BeAFix against ARepair in [2], the authors used AUnit [16] to automatically generate test cases, which resulted in a unusually high rate of incorrect fixes by ARepair. The expectation is that our approach to test case generation is fairer for ARepair. These results are summarized in Fig. 4 for

[9] `https://doi.org/10.5281/zenodo.4676413`

different search depth levels and in the top part of Table 3. ARepair may still report incorrect fixes due to over-fitting; the data considers only correct repairs All executions of TAR were also ran with counterexample-based pruning disabled to answer RQ3. Since feedback is expected to be provided quickly, the timeout was set to 1 minute for all procedures. All tests were run on a Linux-5.15 machine with docker version 20.10 and an Intel Core i5 4460 processor.

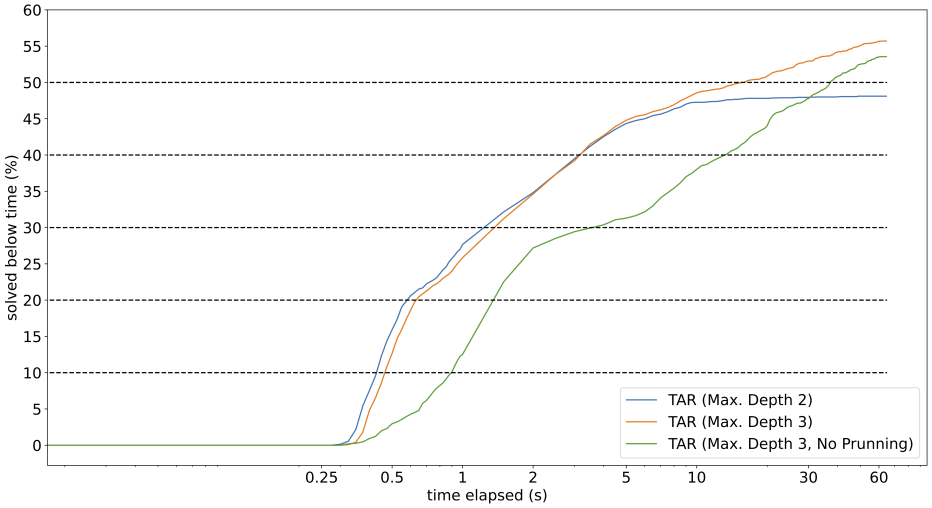

Fig. 3. Percentage of specification challenges repaired by the proposed approach under a certain time threshold, for different search depth levels and with and without pruning

RQ1 The data in Fig. 3 shows that TAR is viable for Alloy 6 repair. It is able to repair about 35% of the specifications by 2 seconds, and by 1 minute it is able to fix 56%, results that even surpass those for non-temporal Alloy repair, as we will shortly see. Increasing the depth to 3 does not seem to increase significantly the performance of the approach, and with depth 2 the results stagnated at 45% by 10 seconds. As shown in Table 3, of the 46% challenges that failed to be fixed under 1 minute, 10% were due to time-out while the remainder failed due to an exhausted search space.

RQ2 As can be seen in Fig. 4, TAR consistently outperforms BeAFix, particularly in smaller time thresholds. At 2 seconds, BeAFix is able to fix 27% of the specifications against the 42% of TAR, a 60% improvement. Although by the 1 minute threshold the difference is reduced to 16%, we consider this to be already too long for a student to wait for automatic feedback. BeAFix was able to successfully fix 47% of the specifications within an 1 hour timeout [2], which is still less than the 52% of TAR with a 1 minute.

Exercise	Cases	ARepair (25 Tests) Fixed (%)	TO	Failed	BeAFix Fixed (%)	TO	Failed	TAR Fixed (%)	TO	Failed
`Classroom`	999	102 (10%)	246	651	311 (31%)	578	110	408 (41%)	52	539
`CV`	137	26 (19%)	6	105	77 (56%)	44	16	85 (62%)	1	51
`Graphs`	283	181 (64%)	0	102	220 (78%)	28	35	240 (85%)	4	39
`LTS`	249	20 (8%)	5	224	35 (14%)	144	70	33 (13%)	5	211
`Production`	61	18 (30%)	1	42	47 (77%)	10	4	50 (82%)	0	11
`Trash`	206	89 (43%)	6	111	182 (88%)	14	10	193 (94%)	0	13
total (static)	1935	436 (22%)	264	1235	872 (45%)	818	245	1009 (52%)	62	864
`TrashLTL`	2890	-	-	-	-	-	-	1832 (63%)	116	942
`Trains`	781	-	-	-	-	-	-	213 (27%)	47	521
total (temporal)	3671	-	-	-	-	-	-	2045 (56%)	163	1463

Table 3. Performance of the 3 techniques under 1 minute threshold and maximum depth 3

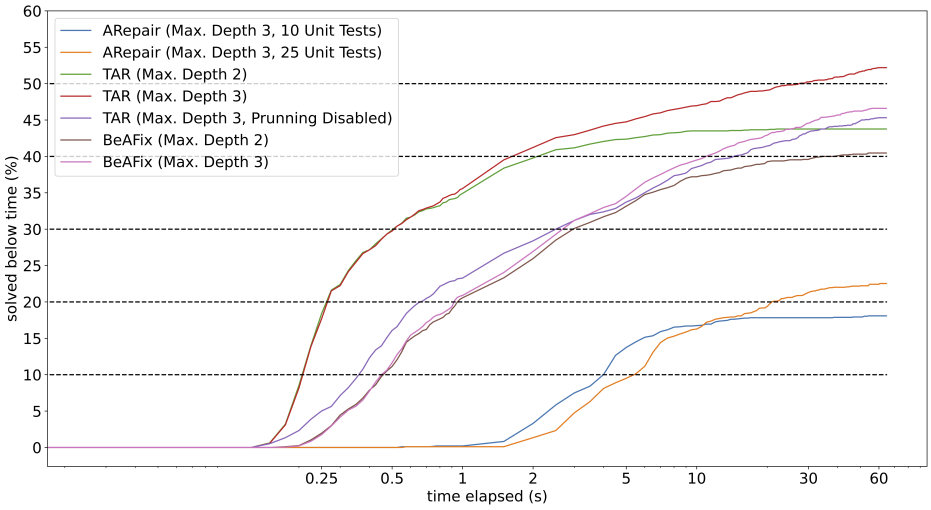

Fig. 4. Percentage of submissions to static challenges correctly repaired by the proposed approach under a certain time threshold, for different search depth levels and with and without pruning

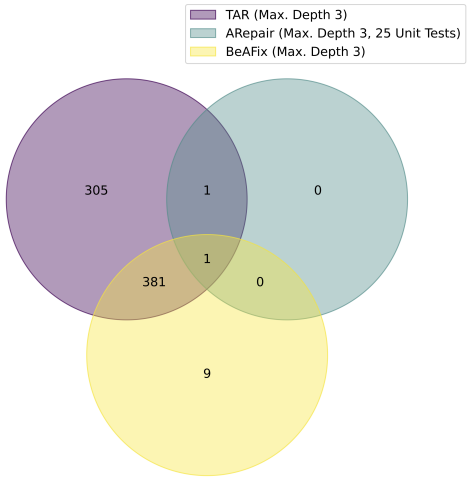

Fig. 5. Classification of submissions to static challenges according to which tool was able to effectively repair them under 1 second

In our evaluation, ARepair was only able to propose a repair that passed the oracle check in less than 5% of the specifications by 2 seconds, and 22% within 1 minute using the top 25 counterexamples selected using TAR's pruning technique. This strategy for the generation of test cases proved to be fairer than the experiments in [2] where ARepair only proposed fixes that passed the oracle for 9% of specifications for the same dataset within 1 hour, but is still well below the performance of TAR and BeAFix. Even disregarding the oracle check, ARepair reported to be able to pass all unit tests in 814 (42%) submissions for the benchmark with 25 unit tests, which is still below TAR and BeAFix.

To provide a better understanding of how the specifications fixed by each of the 3 techniques overlap, the Venn diagram in Fig. 5 classifies the specifications according to which tool was able to repair them under a threshold of 1 second. There was no specification that ARepair was able to repair that TAR missed. As for BeAFix, there were 9 specifications that BeAFix repaired and TAR failed within 1 second, against 306 the outer way around. Of those 9 cases, 6 were due to the fact that BeAFix was able to find a repair under 1 second while TAR took longer. The other 3 required the introduction of a join operation that changed the arity of the relational expression, a mutation supported by BeAFix but not by our InsertBinary.

RQ3 Looking at the performance of TAR with pruning disabled shown in Figs. 3 and 4, it is clear that the impact is particularly relevant at lower threshold levels: at 2 seconds, the technique without pruning is only able to fix about 27% of the specifications, 22% less that the 35% fixed with pruning. As the threshold increases the impact of the pruning technique is reduced. Furthermore, the average amount of generated counterexamples – which amount to calls to

the solver – is low, around 5.4 for the static benchmarks; and around 6.4 for the temporal benchmarks. These counterexamples end up being able to prune an impressive amount of candidates, the number being, on average, around 100000 for the static benchmarks, 76% of which are pruned by the same counterexample; and 160000 for the temporal benchmarks, 67% of which are pruned by the same counterexample.

5 Conclusions

This paper presented a mutation-based technique for the automatic repair of Alloy 6 specifications, being the first to consider its full temporal first-order logic. A new pruning technique was proposed that is suitable for target context of the new technique, namely educational scenarios. Evaluation over a dataset of student submissions has shown that, in this scenario, the proposed technique is able to produce timely repairs and that it outperforms existing approaches to Alloy repair (when considering only the non-temporal Alloy subset).

To be effectively used as a hint system in the educational context, the found repairs must be translated back into a hint that can guide the student in the right direction without explicitly providing the correct solution. The technique for the derivation of hints from repairs, and its subsequent implementation in the Alloy4Fun platform, is the next step in our research plan. This is expected to be followed by proper empirical study to assess the usability and efficacy of the technique in the educational context.

The proposed pruning technique registers the counterexamples that were able to discard the most mutants. Arguably, such counterexamples are more "representative" of the expected property as they identified the most semantically different formulas, a reasoning we followed to use this rank to generate test cases for ARepair in the evaluation. We intend to explore whether this information would also be helpful feedback to the students, namely whether returning the top ranking counterexamples from the pruning process is more productive than the randomly generated ones.

A technique for Alloy repair was developed independently of this work and published after the submission of this manuscript [21]. The timing did not allow for a proper comparison, but we intend to expand our evaluation against it in the short term.

References

1. Brida, S.G., Regis, G., Zheng, G., Bagheri, H., Nguyen, T., Aguirre, N., Frias, M.F.: BeAFix: An automated repair tool for faulty Alloy models. In: ASE. pp. 1213–1217. IEEE (2021)
2. 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. pp. 1135–1147. IEEE (2021)
3. Cai, C., Sun, J., Dobbie, G.: Automatic B-model repair using model checking and machine learning. Autom. Softw. Eng. **26**(3), 653–704 (2019)

4. Clarisó, R., Cabot, J.: Fixing defects in integrity constraints via constraint mutation. In: QUATIC. pp. 74–82. IEEE Computer Society (2018)
5. Edwards, J., Jackson, D., Torlak, E.: A type system for object models. ACM SIGSOFT Software Engineering Notes **29**(6), 189–199 (2004)
6. Gazzola, L., Micucci, D., Mariani, L.: Automatic software repair: A survey. IEEE Trans. Software Eng. **45**(1), 34–67 (2019)
7. Goues, C.L., Pradel, M., Roychoudhury, A.: Automated program repair. Commun. ACM **62**(12), 56–65 (2019)
8. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, revised edn. (2012)
9. Jackson, D.: Alloy: A language and tool for exploring software designs. Commun. ACM **62**(9), 66–76 (2019)
10. Krishnamurthi, S., Nelson, T.: The human in formal methods. In: FM. LNCS, vol. 11800, pp. 3–10. Springer (2019)
11. Macedo, N., Brunel, J., Chemouil, D., Cunha, A., Kuperberg, D.: Lightweight specification and analysis of dynamic systems with rich configurations. In: SIGSOFT FSE. pp. 373–383. ACM (2016)
12. Macedo, N., Cunha, A., Pereira, J., Carvalho, R., Silva, R., Paiva, A.C.R., Ramalho, M.S., Silva, D.C.: Experiences on teaching alloy with an automated assessment platform. Sci. Comput. Program. **211**, 102690 (2021)
13. McBroom, J., Koprinska, I., Yacef, K.: A survey of automated programming hint generation: The HINTS framework. ACM Comput. Surv. **54**(8), 172:1–172:27 (2022)
14. Pei, Y., Furia, C.A., Nordio, M., Meyer, B.: Automatic program repair by fixing contracts. In: FASE. LNCS, vol. 8411, pp. 246–260. Springer (2014)
15. Schmidt, J., Krings, S., Leuschel, M.: Repair and generation of formal models using synthesis. In: IFM. LNCS, vol. 11023, pp. 346–366. Springer (2018)
16. Sullivan, A., Wang, K., Khurshid, S.: AUnit: A test automation tool for Alloy. In: ICST. pp. 398–403. IEEE Computer Society (2018)
17. Toll, D., Wingkvist, A., Ericsson, M.: Current state and next steps on automated hints for students learning to code. In: FIE. pp. 1–5. IEEE (2020)
18. Wang, K., Sullivan, A., Khurshid, S.: Automated model repair for Alloy. In: ASE. pp. 577–588. ACM (2018)
19. Wang, K., Sullivan, A., Khurshid, S.: ARepair: A repair framework for Alloy. In: ICSE (Companion Volume). pp. 103–106. IEEE / ACM (2019)
20. Wang, K., Sullivan, A., Marinov, D., Khurshid, S.: Fault localization for declarative models in Alloy. In: ISSRE. pp. 391–402. IEEE (2020)
21. Zheng, G., Nguyen, T., Brida, S.G., Regis, G., Aguirre, N., Frias, M.F., Bagheri, H.: ATR: Template-based repair for Alloy specifications. In: ISSTA. pp. 666–677. ACM (2022)
22. Zheng, G., Nguyen, T., Brida, S.G., Regis, G., Frias, M.F., Aguirre, N., Bagheri, H.: FLACK: Counterexample-guided fault localization for Alloy models. In: ICSE. pp. 637–648. IEEE (2021)