ATR: Template-Based Repair for Alloy Specifications

Guolong Zheng
University of Nebraska-Lincoln
USA

ThanhVu Nguyen
George Mason University
USA

Simón Gutiérrez Brida
University of Rio Cuarto and CONICET
Argentina

Germán Regis
University of Rio Cuarto
Argentina

Nazareno Aguirre
University of Rio Cuarto and CONICET
Argentina

Marcelo F. Frias
Buenos Aires Institute of Technology and CONICET
Argentina

Hamid Bagheri
University of Nebraska-Lincoln
USA

ABSTRACT

Automatic Program Repair (APR) is a practical research topic that studies techniques to automatically repair programs to fix bugs. Most existing APR techniques are designed for imperative programming languages, such as C and Java, and rely on analyzing correct and incorrect executions of programs to identify and repair suspicious statements.

We introduce a new APR approach for software specifications written in the Alloy declarative language, where specifications are not "executed", but rather converted into logical formulas and analyzed using backend constraint solvers, to find specification instances and counterexamples to assertions. We present ATR, a technique that takes as input an Alloy specification with some violated assertion and returns a repaired specification that satisfies the assertion. The key ideas are (i) analyzing the differences between counterexamples that do not satisfy the assertion and instances that do satisfy the assertion to guide the repair and (ii) generating repair candidates from specific templates and pruning the space of repair candidates using the counterexamples and satisfying instances. Experimental results using existing large Alloy benchmarks show that ATR is effective in generating difficult repairs. ATR repairs 66.3% of 1974 fault specifications, including specification repairs that cannot be handled by existing Alloy repair techniques. ATR and all benchmarks are open-source and available in the following Github repository: https://github.com/guolong-zheng/atmprep.

CCS CONCEPTS

• **Software and its engineering** → **Software defect analysis**; **Error handling and recovery**.

KEYWORDS

Alloy specification, Automatic Program Repair, Template-based Repair and Synthesis, Counterexamples

ACM Reference Format:
Guolong Zheng, ThanhVu Nguyen, Simón Gutiérrez Brida, Germán Regis, Nazareno Aguirre, Marcelo F. Frias, and Hamid Bagheri. 2022. ATR: Template-Based Repair for Alloy Specifications. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), July 18–22, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3533767.3534369

1 INTRODUCTION

Declarative specification languages have been used for diverse software engineering problems. In particular, the Alloy specification language [27], which relies on relational algebra and first-order logic, has been applied to a wide range of software engineering tasks, including software verification [8, 19–21, 44, 50, 55, 60, 70–72], test case generation [32, 45, 49, 58, 63], software design [10–12, 28, 34, 68], network security [43, 52, 61], security analysis of emerging platforms, such as IoT and Android platforms [3, 4, 6, 7, 9, 13, 50, 51, 56, 59, 62], among others. Moreover, the use of Alloy for software specification is tightly integrated with the Alloy Analyzer, allowing users to automatically check that the given specification satisfies some desirable properties. This, in turn, makes the problem of testing and verifying Alloy specifications much easier and directly integrated into the Alloy environment.

Despite the power of the Alloy Analyzer, Alloy users, just like Java or C developers, can introduce subtle bugs when writing specifications. While the Alloy Analyzer can help to automatically check properties and generate counterexamples showing assertion violations, the actual debugging tasks of evaluating and fixing the issues remain challenging and manual, especially for specifications that model large systems and complicated behaviors.

Compared to the rich literature and techniques in automatic program repair (APR) for imperative languages, there are much fewer APR techniques for Alloy. ARepair [74], arguably the first APR work for Alloy, mimics APR techniques that fix bugs witnessed by failing test cases and thus assumes the availability of tests as an oracle for correctness. While tests are common for programming languages, they are not often used in Alloy specification settings, where users write assertions to describe expectations and employ the Alloy Analyzer to check them. In addition, just like in "imperative" APR approaches, ARepair suffers from the *overfitting* problem when tests are inadequate (i.e., the generated repair passes the tests, but is incorrect in general).

In contrast, the recent BeAFix [25] technique directly uses assertions as correctness oracles and reduces the overfitting problem. However, this approach can be slow as it exhaustively enumerates the possible candidate repairs up to a certain bound and checks if any of them successfully satisfies the failing assertions. Indeed, as shown in this work, BeAFix fails to generate complex repairs for many Alloy specifications as those would require trying a prohibitively large number of repair candidates.

In this work, we introduce ATR, a new APR technique and tool that fixes Alloy specifications failing given assertions. Given an Alloy specification and a property that is not satisfied by the specification, ATR first uses an existing Alloy fault localization tool as a blackbox to identify suspicious expressions causing the violation. ATR next queries the underlying Alloy Analyzer for a counterexample, an instance of the specification that does not satisfy the property, and then uses a partial max satisfiability (PMAXSAT) solver [16] to find an instance that does satisfy the property and is as close as possible to the counterexample. ATR then generates repairing expressions from predefined templates and uses the counterexamples and satisfying instances to prune candidate repairs.

Thus, unlike ARepair relies on tests, ATR is similar to BeAFix and uses well-established and widely-used assertions, naturally compatible with the specification practices in Alloy. The main novelty of ATR is that it generates repairs using templates and exploits the differences between counterexamples and satisfying instances of the specification to guide the repair process. These instances are generated by constraint solvers, which are the main underlying technologies in Alloy, and their differences significantly help ATR prune the search space and synthesize much richer and more accurate repairs.

We evaluate ATR on two benchmarks, developed by other research groups, consisting of 1974 buggy specifications from ARepair [73] and Alloy4Fun [42]. The experimental results corroborate that ATR is able to consistently repair faulty Alloy specification with 66.3% repair rate, compared to 9.8% repair rate of ARepair and 50.9% of BeAFix. The average run time to fix each specification is 364.4 seconds, compared to 1569.8 seconds of BeAFix and 91.9 seconds of ARepair (though ARepair also generates many overfitting repairs). The experiments show that ATR can repair nontrivial bugs effectively and in many cases can even synthesize new expressions to complete empty predicates. The repairs generated by ATR are similar to manual repairs, i.e., with short syntax edit distance.

To summarize, our contributions are two-fold. First, we present a novel APR technique for declarative specifications in the Alloy language. The insight underlying our approach is using templates and the counterexamples and closely related satisfying instances to guide and reduce the space of the repair search. Second, we develop a fully automated APR tool called ATR that effectively realizes our APR approach. We evaluate ATR using a large list of buggy Alloy specifications found in prior work and show the effectiveness of ATR on these benchmarks. We make ATR and all experimental results open-source and publicly available on Github: https://github.com/guolong-zheng/atmprep.

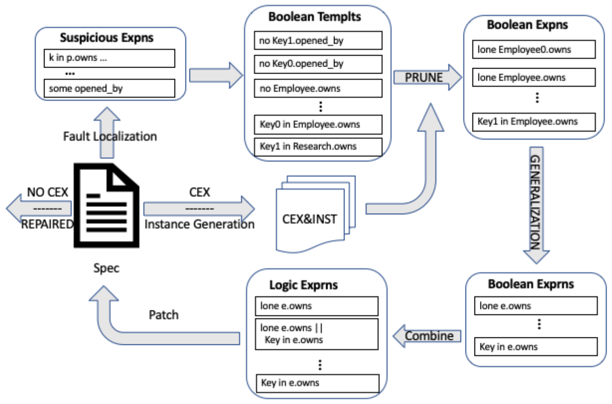

Figure 1: Overview of ATR

2 OVERVIEW

Fig. 1 gives the overview of ATR, which takes as input a specification that violates an asserted property. ATR first invokes a fault localization tool to obtain a list of suspicious expressions. ATR next generates pairs of counterexamples and satisfying instances of the specification concerning the asserted property. To have a more accurate comparison among these instances, ATR uses a PMAXSAT solver to generate satisfying instances that are as similar as the counterexamples as possible.

From the generated instances, ATR then generates a set of boolean templates using atoms, signatures, and fields and prunes these templates based on their results evaluated from the counterexample and satisfying instances. From the remaining templates, ATR generalizes them to Alloy expressions and combines the expressions using logic operators.

Finally, ATR patches the suspicious statements with the generated expressions and checks if Alloy can find counterexamples for the assertion in the modified specification. If Alloy finds no counterexample, then ATR has found a fix. Otherwise, ATR uses the new counterexamples to generate satisfying instances and repeats the repair process.

2.1 Background

Alloy. An Alloy [26] specification captures a formal software model using the following main elements: (1) `sig`s (signatures) are used to define basic data domains, which may be structured using `fields`, defining relationships between `sig`s; (2) `fact`s introduce always-hold constraints (model assumptions) written in Alloy's logic (first-order logic with relational operators, including closures); `fun`s (functions) are named parameterized Alloy expressions, and `pred`s (predicates) are named parameterized Alloy formulas, which can be used to define model operations and properties, as well as for defining facts; and (3) `check` and `run` commands invoke an automated analyzer for checking `assert`s (assertions, Alloy formulas) and running `pred`s, respectively.

Alloy's automated analysis is provided by the Alloy Analyzer, which translates Alloy specifications into propositional formulas and uses underlying SAT solvers to find counterexamples violating the asserted properties (when invoked with `check`), and specification instances satisfying predicates (when invoked with `run`). Intuitively, given a specification M and an assertion p, Alloy Analyzer will search for counterexamples by solving a propositional formula for the constraints of $M \wedge \neg p$; in contrast, when p is a predicate, Alloy Analyzer will search for instances satisfying p by solving $M \wedge p$. Alloy's analysis is bounded-exhaustive, i.e., given a specification M, an assertion (resp. a predicate) p, and a bound k (called the *scope* in the context of Alloy), the Alloy Analyzer will find a counterexample (resp. satisfying instance) for p of size bounded by k, if and only if such k-bounded counterexample (resp. instance) exists.

Fault Localization. One of the main components of an APR technique is fault localization, which identifies suspicious statements in the program where the error occurs. An effective fault localization algorithm can significantly reduce the search space and guide the repair process to the relevant code region.

The current implementation of ATR uses the FLACK fault localization tool for Alloy [77]. FLACK takes as input an Alloy specification violating some assertion and returns a list of suspicious expressions likely contributing to the violation. Because ATR treats fault localization tool as a blackbox, it can use other Alloy fault localization tools such as AlloyFL [75].

Partial Maximum Satisfiability. To synthesize and prune relevant candidate repairs, ATR searches for an instance that satisfies the given specification but is closely similar to the counterexample. To do this, we reduce this task to a PMAXSAT (partial maximum satisfiability) problem [39] and solve it using Pardinus [16].

The PMAXSAT problem is defined as follows: given a set of hard clauses and a set of soft clauses, find a solution that satisfies *all* the hard clauses and *as many soft clauses* as possible. ATR encodes the specification M and the property p as the hard constraints and the counterexample as the soft constraints, and finds an instance that satisfies p and is as similar to the counterexample as possible.

2.2 Illustrative Example

Fig. 2 shows an Alloy specification for a room access control problem. First, we define several classes or types (*sig*'s): `Room` has one subclass `SecureLab`; `Person` has two non-overlapping subclasses `Employee` and `Researcher` sharing a common field `owns` that represents a set of keys owned by a person; and `Key` has a field `opened_by` representing exactly one Room opened by the key. We also define a couple of *facts* capturing additional constraints. The first specifies that there are at least one employee and one researcher, at least one key to open `SecureLab`, and some employee owns more than one key.

The second specifies that every room can be opened by some key, and each employee owns some key and this key can not open `SecureLab`. Next, we have a *predicate* `CanEnter`, a boolean function checking that a person can enter a room only if they own a key to that room. Finally, to ensure the secure access of `SecureLab`, we use the *assert* `is_secured` to check that only researchers can enter the secure lab.

```
1  sig Room {}
2  one sig SecureLab extends Room {}
3
4  abstract sig Person { owns : set Key }
5  sig Employee extends Person {}
6  sig Researcher extends Person {}
7
8  sig Key { opened_by: one Room }
9
10 fact {
11   some Employee && some Researcher
12   some e : Employee | #e.owns > 1
13 }
14
15 fact {
16   all r : Room | some opened_by.r
17   all e : Employee | some k : Key | k in e.owns
18   and SecureLab != k.opened_by  //bug
19   //and e !in owns.opened_by.SecureLab //missing constraint
20 }
21
22 pred CanEnter(p: Person, r:Room) {r in p.owns.opened_by}
23
24 assert is_secured {
25   all p : Person | CanEnter[p, SecureLab]
26     implies p in Researcher
27 }
28 check is_secured
```

Figure 2: A buggy Alloy specification.

To check the assertion, we run the Alloy Analyzer on this specification with the `check` command as shown on line 28. The Analyzer disproves the assertion by generating a counterexample, shown in Fig. 3, which describes a scenario in which an employee owns a key that opens `SecureLab`. Thus, the specification contains a bug violating the asserted property. Our goal is then to use ATR to fix the bug.

Fault Localization. Similar to most APR approaches, ATR first identifies where the bug occurs. By using the Alloy fault localization tool FLACK [77], we find that the fault is likely caused by the fact on line 18. This fact specifies that every employee owns some key that can not open SecureLab. However, this fact misses a constraint that an employee can not own keys that can open the secure lab.

To repair this "under-constraint" defect, ATR needs to synthesize this new constraint and add it to the expression on line 18. We note that none of the existing Alloy repair techniques can fix this specification violation. For example, both ARepair [74] and BeAFix [25] tools indicate that they fail to repair the violation.

Counterexample and Satisfying Instances. ATR exploits the rich details captured by instances generated by the Alloy Analyzer. To synthesize a repair for the suspicious expression returned by fault localization, ATR analyses the differences between instances that *do not* satisfy the assertion (i.e., counterexamples) and those that *do* satisfy the assertion (i.e., satisfying instances) to reduce the search space.

The Alloy Analyzer has already provided the needed counterexample violating the assertion, e.g., Fig. 3. However, we still need a satisfying instance. One way to find such an instance is creating a predicate `pred is_secured` with similar contents as those from `assert is_secured` and asking the Alloy Analyzer to find an instance, such as the one given in Fig. 4, that satisfies the predicate.

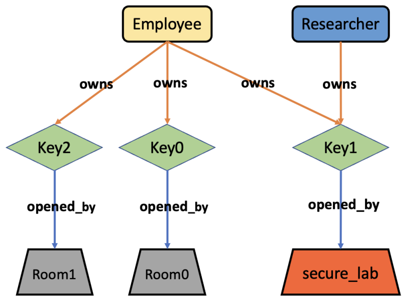

Figure 3: Counterexample

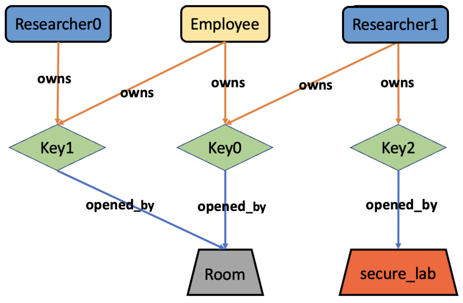

Figure 4: Random SAT instance

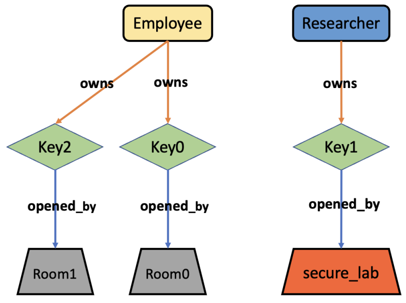

Figure 5: PMAXSAT instance

We hypothesize that the differences between the counterexample and satisfying instances can help debug the issue. However, in this case, the counterexample in Fig. 3 and the satisfying instance in Fig. 4 are too different to be useful. For example, `Researcher0` owns a `Key1` that opens `Room` in the satisfying instance but does not own any key in the counterexample. However, this difference is unrelated to the defect. Indeed, satisfying instances randomly generated from the Alloy Analyzer often give many irrelevant differences to the counterexamples.

We need counterexamples and satisfying instances that have as *few differences* as possible. To obtain this, we treat the problem as a PMAXSAT problem, in which we set the counterexample as soft constraints and the Alloy specification and the predicate as hard constraints, and then use a PMAXSAT solver to find an instance that satisfies all the hard constraints (i.e., satisfying the specification) and as many soft constraints as possible (i.e., similar to the counterexample).

In this example, using the Pardinus [16] PMAXSAT solver, ATR obtains the satisfying instance in Fig. 5 that is very similar to the counterexample in Fig. 3. The only difference here is that `Employee` does not own `Key1` that opens the secure lab. This difference is precisely the property causing the counterexample to violate the assertion.

Candidate Repairs. Unlike a traditional program (e.g., C), composed of multiple simple statements, Alloy has fewer but more complicated expressions. Thus, the search space is more extensive as we need to consider candidate fixes for the entire expression on line 18 and its subexpressions.

ATR uses the details from the counterexample and instances to generate relevant expressions to repair the buggy one on line 18. Moreover, to further reduce the search space, we generate new expressions under a specific grammar (template) consisting of commonly used Alloy binary and unary boolean and relational operators (given in Fig. 6 and described in § 3.3).

Here, ATR uses seven atoms `Employee0`, `Researcher0`, `Key0`, `Key1`, `Room0`, `Room1` and `SecureLab0`, which are atoms shared between the counterexample and the satisfying instance; five sigs `Employee`, `Researcher`, `Key`, `Room` and `SecureLab`; and two fields `owns` and `opened_by`, that are collected from all defined sigs and fields in the specificaiton. From these, ATR uses the relational operators in line 2 in Fig. 6 to generate relational terms such as `Key0.opened_by`, `Employee + Researcher`, `Employee.*owns`, and combines them with boolean operators in line 7 to form expressions such as `no Key0.opened_by`, `some Employee.*owns`, `Key1 in Researcher.owns`.

Note that the generated expressions (e.g., `Key1 in Employee.o wns`) contains atoms (e.g., `Key1`) that are specific to the generated instances and are not valid Alloy expressions. Thus, when patching repairs, ATR generalizes these to live variables or relation terms in the specification scope (e.g., $Key1$ becomes k or Key).

Pruning. Depending on the differences and the defect, we could generate many possible repair candidates. However, we can prune candidate expressions that give the same results when evaluating against the counterexample and satisfying instances. This is because while the counterexample and satisfying instances are similar by design, they inherently have different semantics (one does and the other does not satisfy the instance) and must give opposite results.

For example, the candidate expression `no Key0.opend_by` is pruned because it evaluates to False for both instances in Figs. 3 and 5. In contrast, `Key1 in Employee.owns` is kept as it evaluates to False in the satisfying instance and True in the counterexample. In our experience, this step helps reduce the search space significantly.

Synthesizing Repairs. ATR replaces the buggy expression with each candidate and checks that the Alloy analyzer cannot find any counterexample for the new specification (i.e., the specification does not violate the assertion). Note that ATR also requires the Alloy analyzer to find some satisfying instance for the new specification (otherwise this means the specification is too strong and not satisfiable). If the Alloy Analyzer found a satisfying instance and no counterexample, we consider the new specification as a valid fix. Otherwise, ATR repeats the process with the old and new counterexamples.

Note that if the original specification has other predicates and assertions other than the violated assertion, then the new specification, which fixes the violation, also preserves similar behaviors for those predicates and assertions as to the original one. This is similar to the positive and negative tests in traditional APR, where a repair candidate has to pass both the existing positive tests and the negative ones.

For this example, ATR finds the following repair:

```
all e : Employee | some k : Key | k in e.owns and
    SecureLab != k.opened_by and
    e !in owns.opened_by.SecureLab // added constraint
```

Table 1: Mutation Operations

MO	Description	Example
UOM	Unary Operator Modification	`*a` ↦ `~a`
UOD	Unary Operator Deletion	`*a` ↦ `a`
UOA	Unary Operator Addition	`a` ↦ `*a`
BOM	Binary Operator Modification	`a + b` ↦ `a - b`
BOD	Binary Operand Deletion	`a + b` ↦ `b`
BOS	Binary Operand Swap	`a in b` ↦ `b in a`
LOM	List Operator Modification	`a&&b&&c` ↦ `a||b||c`
LOD	List Operand Deletion	`a||b||c` ↦ `a||c`

which adds a new constraint requiring the keys to open the secure lab are not owned by any employee.

Complex Repairs and Program Synthesis. For demonstration purposes, the above example is relatively simple and thus allows ATR to generate the repair in the first iteration. However, for more complicated cases, ATR might need multiple iterations to find the fix. For example, the specification `classroom_inv13_32.als` in our experiment requires two iterations to generate the expression `Tutors.Person in Teacher && Person.Tutors in Student` to fix the bug `no Student.Tutors`.

Note that ATR can also be used as a synthesis tool that generates missing expressions in an Alloy specification. This is useful because many Alloy violations are due to missing information or facts. For example, the `bst`(binary search tree) example in our benchmark has an empty predicate, and ATR was able to synthesize the expression `no n.left || no n.right` to it (in 2 iterations) and thus completes the specification.

3 THE ATR APPROACH

In § 2 we give an overview of ATR in Fig. 1 and show how ATR works using a concrete example. In this section we describe the main ideas and designs of ATR.

3.1 Preprocess with Mutation

There are many bugs that can be repaired by simple mutations, for example, in *addr.als* of ARepair benchmark, the buggy expression `all b : Book | n : Name | lone b.n.listed` can be repaired by simply changing `lone` to `some`. To rapidly repair those simple bugs and generate initial counterexamples, ATR mutates the buggy expressions before starting template-based searching.

ATR uses eight mutation operations defined in Table 1: UOM (Unary Operator Modification) that changes a unary operator; UOD (Unary Operator Deletion) that deletes a unary operator; UOA (Unary Operator Addition) that adds a unary operator; BOM (Binary Operator Modification) that changes a binary operator; BOD (Binary Operand Deletion) that deletes a binary operand; BOS (Binary Operand Swap) that swaps the position of two operands if the binary operator is asymmetric; LOM (List Operator Modification) that changes a list operator; and LOD (List Operand Deletion) that deletes a list operand).

Given an expression, ATR applies mutation operators multiple times upto a certain bound, e.g., `a in b` becomes `b in ~a` using 3 mutations: BOS, UOD, and UOA.

For each mutant, ATR checks if the Alloy analyzer generates any counterexamples. If no counterexample is generated, ATR considers this mutant as a repair and returns; otherwise, ATR checks if previous counterexamples are also counterexamples for the mutant. If all previous counterexamples satisfy the mutant, i.e., not counterexamples of the mutant, ATR considers the mutant a different mutant and asks the Alloy analyzer to generate a counterexample and adds it to the initial counterexample suite; otherwise, ATR considers the mutant the same as previous mutants and continues.

3.2 Instance Generation

Given a counterexample and an assertion, ATR generates one single instance that satisfies the asserted property with minimal differences from the counterexample. The reason for this is that while every counterexample and every satisfying instance will be different, we want pairs with the fewest differences possible so that we can analyze them effectively.

Thus, to obtain the desired satisfying instance that is as close to the counterexample as possible, ATR treats this problem as a PMAXSAT problem as described in § 2.1. More specifically, ATR converts the assertion to a predicate, and sets the predicate with the Alloy specification as the hard constraints, and the counterexample as the soft constraints. In this work, ATR uses the Pardinus [16] solver to solve the PMAXSAT problem and returns an instance that satisfies the predicate and has minimum differences with the counterexample.

As demonstrated in § 2.2, the reason to generate instances with minimal differences is to narrow the search space for candidate repair. While we can use the Alloy Analyzer to generate a satisfying instance, such randomly generated instances are quite different from the counterexample (cf. Fig. 3 and Fig. 4). More specifically, such randomly generated instances often contain properties that are irrelevant to the defect, hindering the discovery of properties related to the defect. Besides, the names of atoms in the Alloy analyzer are also randomly assigned, which may lead to mismatching between shared atoms and misleading the search process.

For example, considering the counterexample in Fig. 3 and the random Alloy generated instance in Fig. 4, the difference shows that `Researcher0` owns `Key1` that opens `Room`, which does not affect the satisfiablity of the counterexample and the satisfying instance, and it may lead to over-constraint repairs that preventing `Researcher` to own `Key` to `Room`. With the PMAXSAT generated instance in Fig. 5, those unrelated differences are pruned out, and ATR can easily identify the crucial difference that `Employee` should not own key to `SecureLab`.

3.3 Generating and Pruning Templates

ATR generates repair candidates by enumerating candidate templates using the grammar given in Fig. 6. There are three different templates: relational templates (or terms) that connect variables using relational operators, shown in lines 5–7; boolean templates that connect relational expressions using boolean operators, shown in lines 1–3; and logic templates that connect templates using logic operators, as shown in lines 9–11.

Given a pair of counterexample and satisfying instances, ATR first generates all possible relational terms using shared atoms,

```
<RT>  := <V> | <RUO> <RT> | <RT> <RBO> <RT>
<RUO> := ~ | * | ^                   <RBO> := . | & | + | -
<v>   := atoms + sigs + fields

<BT>  := <BUT> | <BBT>
<BUT> := <BUO> | <RT>                <BBT> := <RT> <BBO> <RT>
<BUO> := no | lone | some | one      <BBO> := = | in | != | !in

<LT>  := <LUT> | <LBT> | <BT>
<LUT> := <LUO> <LT>                  <LBT> := <LT> <LBO> <LT>
<LUO> := !                           <LBO> := || | && | => | <=>
```

Figure 6: Repair Templates

sigs, and fields up to a certain bound by the number of relational operators. For example, `Employee.*owns.Key1` is generated using sig `Employ`, field `owns` and atom `Key1`, with a bound of three operators: two dot join (`.`) operators and one transitive closure `*` operator.

Next, ATR generates boolean expressions by filling the holes in the boolean templates with the generated relational expressions. For example, given two terms `Employee.owns` and `Key`, for a asymmetric template `__ in __`, two expressions are generated: `Employee.owns in Key` and `Key in Employee.owns`. For a symmetric template `__ = __`, only one expressions is generated: `Employee.owns = Key`.

Pruning. When filling the holes, instead of using concrete relational terms, ATR uses values to fill the holes in the templates. Given an instance, many terms evaluate to same value and are semantically equivalent in that context. For example, `Key.opened_by` and `Room` are considered equivalent in Fig. 3 because they both evaluate to value `Room0 + secure_lab0`.

If a term evaluates to the same value in the counterexample and satisfying instance, we use that value to represent the term; otherwise, we use the term. For example, `some Key.opened_by` and `some Room` become `some Room0+secure_lab0` because `Key.opened_by` and `Room` both evaluate to `Room0+secure_lab0`. Whereas `some owns` cannot be represented by value as `owns` evaluates to `Key0+Key1` in the counterexample and `Key0` in the satisfying instance.

The similarity introduced by the PMAXSAT helps to reduce the number of different values significantly. For the instances in Fig. 3 and Fig. 5, there are 2855 relational terms, but only 136 different values and 23 different terms, which reduces the search space of each hole from 2855 to 159.

ATR then prunes the generated expressions based on the counterexample and the satisfying instance. ATR evaluates each expression in the context of the counterexample and the satisfying instance and keeps the ones with different evaluation results, i.e, True in the counterexample and False in the satisfying instance, and vice versa. If we can trust the counterexample and the satisfying instance, i.e, the satisfying instance is the expected instance, we can select the expressions evaluating False in the counterexample and True in the satisfying instance. However, the satisfying instances are generated from a "Faulty" specification and may not be an expected instance; we thus need to consider more situations.

Those different expressions are potential properties that differentiate between P and $\neg P$ and can help repair the defects. While the expressions with the same evaluation results are pruned out, as they are most likely implied by the bug-free part of the specification, and may not relate to the defect.

3.4 Synthesizing Repairs

The boolean expressions contain atoms that are bounded to the specific pair of counterexample and satisfying instance. ATR generalizes those boolean expressions to valid Alloy expressions by replacing the atoms with live variables at the buggy location. At the buggy line 18 in Fig. 2, we collect 2 live variables of type `Key`: `k` and `Key`, and `Key1 in Employee.owns` can generalize to `k in Employee.owns` or `Key in Employee.owns`.

ATR then combines the boolean expressions using logic operators to generate complete expressions. For example, the expression `k in Employee.owns => some owns` is generated by connecting `k in Employee.owns` and `some owns` with `implies`.

ATR patch the buggy specification with these expressions in three ways. For example, given the buggy expression:

```
all e : Employee | some k : Key | k in e.owns
  and SecureLab != k.opened_by
```

and the synthesized expression `opened_by.SecureLab !in e.owns`, ATR generates the following patches:

(1) Connecting with the buggy expression using logic operator.

```
all e : Employee | some k : Key | k in e.owns &&
    SecureLab != k.opened_by implies
    opened_by.SecureLab !in e.owns
```

(2) Replacing part of the buggy expression.

```
all e : Employee | some k : Key | k in e.owns &&
    opened_by.SecureLab !in e.owns
```

(3) Replacing the whole buggy expression.

```
all e : Employee | some k : Key |
    opened_by.SecureLab !in e.owns
```

To ensure the repair is as similar as the original expression and to avoid overconstraint as much as possible, ATR applies these patches in the following order: first try (1), then (2), and finally (3).

Finally, for each new specification, ATR checks if the Alloy analyzer generates any counterexamples and any satisfying instance. If no counterexample is generated, ATR considers that the repair is done and returns; otherwise, ATR generates its corresponding satisfying instance and checks if the pair already exists, if so, the patched expression has no impact on the defect, thus is abandoned and the corresponding expression is removed in the next iteration; otherwise, the patch is saved, and ATR repeats the process with this patched version and the new pair of a counterexample and a satisfying instance.

4 EVALUATION

ATR is implemented in Java. It builds on top of Alloy 4.1 and uses the Pardinus [16] PMAXSAT solver. The tool takes as input an Alloy specification that fails an assertion and returns a specification satisfying that assertion while preserving other behaviors of the original specification (i.e., satisfying other existing assertions and predicates in the specification).

To evaluate ATR, we consider three research questions:

(1) RQ1: Can ATR repair Alloy bugs effectively?
(2) RQ2: What are some interesting characteristics of ATR repairs?

(3) RQ3: How does ATR compare to other Alloy repair techniques?

To investigate these questions, we evaluate ATR using two different benchmarks shown in Tab. 2. The first one is from the Alloy4Fun project [42] and consists of 1936 buggy specifications collected from student submissions for six Alloy problems (*classroom* models class registrations, *cv* models work and source distribution problem, *graphs* models different graph properties, such as acyclic and complete, *lts* models a labeled transition system, *production* models an automated production line, and *trash* models a file system trash can).

The second benchmark is from the ARepair project [73] and consists of 38 buggy specifications collected from 12 Alloy problems: six from the Alloy Analyzer (*addr* models an address book, *cd* models object and class hierarchy, *ctree* models undirected trees, *farmer* models the chicken crossing problem, *bempl* and *other* model the security lab access problem, and *grade* is a gradebook specification) and the rest from graduate student homework (*arr* models sorted arrays, *blancedBST* models balanced binary search tree, *dll* doubly linked lists, *fsm* models finite state machines, and *student* models sorted linked lists).

The specifications are relatively small, i.e., tens or hundreds lines of code. However, they contain real bugs written by humans and consist of a wide variety of defects, from simple ones that can be repaired by modifying a single operator to complicated ones that require synthesizing new expressions and replacing the whole predicate body. Moreover, the correct versions of these specifications are provided with the benchmarks and used as ground truths to check our results.

For our experiments, we use ATR's default setting. We set 5 as the maximum depth of relational expressions, i.e., generated relational expressions contain at most 5 relational operators. We set 3 as the maximum depth of logic structures, i.e., generated expressions contain at most 3 logic operators. For each specification, we use a 1-hour timeout, as used in BeAFix [25]. [1] All these parameters can be modified by the user, and we choose this configuration based on experience.

All experiments reported here were run on a 2.4GHz Intel Xeon CPU with 16GB of RAM with Ubuntu 16.04. ATR and all benchmarks are open-source and available in the following Github repository: https://github.com/guolong-zheng/atmprep.

4.1 RQ1: Performance of ATR

Tab. 2 presents our results. Columns **Models** and **#specs** show the model groups and the number of buggy specifications in the groups. Column **#repair** shows the total number of correctly repaired specifications. Column **#dist** shows the average syntactic distance between the repaired expression and the manual repaired expression. Column **overfit** shows the number of fixes that are overfitting (satisfies the given requirements but incorrect in general). Finally, column **time** shows the average time in seconds, which includes both fault localization and repair time. Note that the average fault localization time is negligible, i.e., less than 1 second.

[1] ARepair used a 15-hour timeout, but its benchmark only contains 38 specifications. Due to the large size of our benchmarks, i.e., 1974 specifications, we choose a 1-hour timeout as BeAFix.

Table 2: ATR's Results on the ARepair and Alloy4Fun benchmarks

	Model	#specs	#repair	dist	overfit	time(s)
Alloy4Fun	classroom	999	688	4	0	601.8
	cv	138	38	8	0	376.2
	graphs	283	260	5	0	156.9
	lts	249	70	4	0	487.3
	production	61	43	6	0	332.8
	trash	206	187	3	0	254.6
	Summary	**1936**	**1286**	**5**	**0**	**368.3**
ARepair	addr	1	1	1	0	0.7
	arr	2	1	1	0	1.9
	balancedBST	3	1	4	1	584.9
	bempl	1	1	1	1	3.3
	cd	2	2	2	0	92.2
	ctree	1	0	-	0	3600
	dll	4	2	4	0	1615.2
	farmer	1	0	-	0	3600
	fsm	2	2	3	1	3.9
	grade	1	1	5	1	1.8
	other	1	1	7	1	2.9
	student	19	10	4	3	443.6
	Summary	**38**	**22**	**3**	**8**	**829.2**
	Total	**1974**	**1308**	**5**	**8**	**364.4**

Successes. ATR was able to repair 1308/1974 (66.3%) specifications using an average time of 364.4s. For the Alloy4Fun benchmark, ATR repairs 1286/1936 (66.4%) specifications using 368.3s on average. For the ARepair benchmark, ATR repairs 22/38 (57.9%) specifications using 829.2s on average.

ATR can repair various kinds of bugs, from simple ones that can be repaired by a single mutation to complicated ones that require multiple templates. For example, in `addr_1.als`, ATR uses just one unary operator modification mutation to change the buggy expression `all b:Book | all n:b.entry | lone b.listed[n]` to `all b:Book | all n:b.entry | some b.listed[n]` to satisfies the assertion that all entries have at least one listed item. For the more complicated specification `classroom_inv13_14.als`, ATR changes the buggy expression `all t:Teacher,s:Student | t->s in Tutors` (all teacher tutors all student) to `Tutors.Person in Teacher && Person.Tutors in Student` (a person tutors must be a teacher and a person tutored must be a student) using the binary template `__ in __` twice and the logic template `__ && __` once.

ATR can also repair defects that cannot be repaired by BeAFix and ARepair. For example, in `classroom_inv12_61`, the user expects that each teacher teaches some groups in some class. BeAFix and ARepair both report no repair found for the buggy expression `all t:Teacher | some g:Group | t.Tutors in g.~(Class.Groups)`, which specifies that each teacher tutors some students without constraints over the teach relation. ATR generates the following fix:

```
all t:Teacher | some (t.Teaches).Groups
```

which specifies that each teacher teaches some groups and therefore satisfies the user's intention.

Fails. ATR was not able to repair 666/1974 (33.7%) specifications. This is due to the inaccuracy of fault localization and the complexity of the bugs (responsible for 556 fails).

First, inaccurate results from FLACK cause ATR to fail to fix 110 specifications. Just like most APR approaches, ATR might not be effective if its fault localization tool does not give accurate results. For example, in `student19.als`, FLACK ranks the buggy expression at 8th place of the ranking list, and ATR times out for this specification. If we manually specify the buggy location for ATR, it finds the correct fix that is the same as the ground truth using 37 seconds. Thus, a more accurate underlying fault localization tool will significantly improve the ATR's performance.

Second, ATR failed to fix 556 specifications due to the complexity of the required repairs. One common failure we observe is that many repairs require introducing new variables. For example, in `cv_inv1_13.als`, the ground truth fix for the buggy expression `some User.visible` is `all u:User | u.visible in u.profile`, which introduces and uses a new qualifer "u". ATR does not consider templates that introduce such arbitrary qualifiers and therefore cannot find such repairs. We can extend ATR for this, e.g., by introducing a new template `all u : __1 | __2 in __3`. However, the introduction of new variables expands the search space exponentially, e.g., `__1` and `__2`/`__3` have 5790 and 17370 possible expressions, respectively, and thus this template alone has $1.7e^{12}$ possible candidates.

4.2 RQ2: Repair Analysis

Similarity to ground truths. ATR's 1308 repairs are similar to the ground truth specifications. On average they are 5 syntactic edits from the ground truths, i.e., specifications generated by ATR can be transformed to the ground truths with 5 edits, including deleting, adding, and replacing nodes on the AST's. For example, in `lts_inv5_16.als`, ATR transforms the buggy expression `all s1,s2:State | s1.trans = s2.trans` to `all s1,s2:State | State.trans.State = s2.trans.State` using one binary boolean template `? = ?`, which is 1 edits away from the manual fix `all s1,s2:State | s1.trans.State = s2.trans.State`, i.e., replacing `s1` by `State`.

We manually analyze the buggy and ATR's patched expressions and find that the ATR's patches do not depend on the syntactic complexity of the buggy expression or the similarity between the buggy expression and the ground truth. Instead, they mainly depend on the syntactic complexity of the correct expression, i.e., the number of operators and variables involved in the expression. Thus, ATR is able to generate fixes even if the buggy expressions are largely different from the correct ones. For example, in `Trash_inv7_52.als`, the user writes `~link.link in iden` to specify that no file links to `Trash`, which underconstrains and allows a `Trash` file links to itself. ATR generates the fix `Trash in (File - File.link)` which specifies that `Trash` are files that no file links to and is semantically equivalent to the ground truth.

For mutation-based techniques, this fix requires at least five mutation operations. Both ARepair and BeAFix fail to find this fix within their time limits. ATR uses templates to replace the buggy expression, and is able to find this fix quickly using 78s (in the first iteration).

A Synthesis Tool. In addition to being an APR tool, ATR can also be used as a stand-alone *synthesis* tool that infers missing expressions in an Alloy specification. This is useful because many Alloy problems are due to missing information or facts (many of the bugs in the benchmarks we used are incomplete specifications.)

For example, the `bst` specification in ARepair benchmark contains an empty predicate:

```
pred HasAtMostOneChild( n : Node ) {}
```

which violates an assertion specifying that a node has at most a child node. For this bug, ATR synthesizes the expression `no n.left || no n.right` specifying that a node `n` has either no left child or no right child in the predicate and thus completes the specification.

The `dll` specification in ARepair benchmark also has an empty predicate `ConsistentPreAndNxt`, which trying to specify that in a doubly linked list, the next node's previous node is itself. ATR completed the specification by synthesizing and inserting the expression `nxt = ~pre`, which specifies that relation `nxt` is equivalent to the inverse of relation `pre`.

Pruning. The ability to scale a large search space of candidate repairs is crucial for any APR and synthesis approaches. ATR employs several tactics to prune repairs such as analyzing counterexamples and satisfying instances and only generating repairs under certain templates as described in § 3.

To show how pruning helps ATR, we ran ATR on the same benchmark with pruning disabled. Unsurprisingly, we found that ATR's performance significantly degrades, in both increasing running time and producing much fewer fixes, when we do not use pruning. More specifically, ATR with pruning repairs 629 more specifications and saves about 15 mins of runtime on average. In short, we found that pruning is effective (in fact, required) in reducing the search space and helps ATR find correct fixes quickly and in general repair more specifications in the given time limit.

Overfitting. Overfitting is a major issue in APR where the fixes pass the given "oracles" (e.g., tests), but are incorrect in general. Assertions typically are an ideal oracle because they can capture a large set of tests. Nonetheless, if the assertions are not sufficiently strong, APR tools including both BeAFix and ATR can still generate overfitting fixes.

ATR produces overfitting fixes for 8/38 ARepair specifications and *zero* Alloy4Fun specifications. For ATR, these fixes are "correct" where the Alloy analyzer finds no counterexamples and some satisfying instances. However, they are incorrect in general as the patched specifications are not semantically equivalent to the ground truth specifications.

We found that ATR's overfitting fixes are mostly caused by inaccurate fault localization, where the bug happens in a predicate, but the fault localization tool reports expressions in a fact. For example, the assertion in `balancedBST3_2.als` specifies that for a balanced tree, the height of a node's left subtree and the height of its right subtree differs at most one. The buggy expression occurs in the predicate `Balanced`, but the FLACK localization tool reports a spurious location in one of the facts. Surprisingly, ATR still can generate a repair candidate that satisfies the assertion (i.e., passes the oracles). Nonetheless, the generated fix, which forces that all nodes must have exactly one parent, is too strong and is thus incorrect in general. If we provide ATR with the correct buggy location in `balancedBST3_2.als`, it generates a correct fix that is semantically equivalent to the ground truth.

Note that other Alloy repair tools such as ARepair and BeAFix can also face overfitting issues. As described next in § 4.3), ARepair is more prone to overfitting as it relies on tests. BeAFix does not generate overfitting patches because it requires the user to provide suspicious locations. If we provide BeAFix with the same inaccurate fault localization information, it would also generate an overfitting repair for the `balancedBST2_3.als` specification.

4.3 RQ3: Comparing Alloy Repair tools

We compare ATR with ARepair [74] and BeAFix [25], the two state of the art Alloy repair tools we are aware of. Similar to ATR, BeAFix uses assertions and thus we reuse the same assertions from BeAFix's experiments [25]. ARepair uses Alloy unit tests instead of assertions as oracles, and we use AUnit [69] to automatically generate such tests to represent assertions (the same way ARepair uses to repair assertion violations [74]). To better evaluate the automatic repair ability, we did not consider additional tests that the ARepair authors manually added. Consequently, our experimental results might be different from those reported in [74]. We use the best performance setting of ARepair as reported in [74]. We set a 1-hour timeout for all tools, the same time limitation used in the experiment of both ARepair and BeAFix.

Tab. 3 presents the results. Column **correct** shows that total number of correct repaired specifications. Column **overfit** shows the total number of overfitting patches [2]. Column **time** shows the average time in seconds.

In summary, for 1974 specifications, ARepair generates patches for 1694/1974 (86%) specifications using an average time of 91.9 seconds. However, 1500 (88.5%) of 1694 generated patches are overfitting fixes. As observed in [25], the overfitting is caused by the limitation of using test suites as oracles and can be improved by providing ARepair with stronger test suites.

BeAFix generates fixes for 1005/1974 (50.9%) specifications with no overfitting using an average time of 1569.8 seconds. ATR generates correct fixes for 1300/1974 (67.1%) specifications and 8 overfitting patches using an average time of 364.4 seconds. Note that BeAFix requires the user to specify suspicious statements to repair, and if such information is inaccurate, BeAFix also generates overfitting repairs as discussed in § 4.2.

ATR shows the best repair rate, 67.1% against 9.8% of ARepair and 50.9% of BeAFix. Manually checking the repaired cases for all tools, we find that ARepair and BeAFix complement ATR, i.e., there are cases ARepair and BeAFix can repair but not ATR, and vice versa. For example, ARepair and BeAFix correctly repair `arr1.als` and `fsm1.als`, that could not be repaired by ATR. [3] In contrast, ATR repairs `lts_inv1_14.als` and `graphs_oriented_9.als` that could not be repaired by ARepair and BeAFix. Thus, the mutation-based approaches ARepair and BeAFix complement our template-based ATR approach.

[2]Overfitting patches pass all tests but are not equivalent to the correct specifications.
[3]For `arr1.als`, the repair needs a template not considered by ATR. For `fsm1.als`, ATR failed to generate a similar counterexample and satisfying instance due to the overconstraint of the bug.

4.4 Threats to Validity

The main threat to external validity is the generalizability of the benchmarks used in the evaluation, i.e., the Alloy specifications and bugs in the benchmarks may not represent general cases. The two benchmarks used in our evaluation have been used to evaluate other Alloy repair tools. All the bugs in both benchmarks are not synthetic but introduced by humans in real-life usage. The benchmarks are admittedly consisting of small Alloy specifications, and may not capture the complexity of large specifications. However, they do involve some complex kernels characteristic of larger realistic Alloy specifications.

ATR builds on top of multiple off-the-shelf tools, e.g., Alloy and its underlying SAT4J solver, and the Pardinus solver. These tools also include some degree of randomness. We run our experiment several times and notice negligible impacts of the solver's nondeterminism.

5 RELATED WORK

Program Repair and Synthesis. APR techniques have steadily gained research interests and produced many novel repair techniques. *Constraint-based* repair approaches, e.g., Angelix [47], AFix [29], SemFix [53], FoRenSiC [15], StarFix [76], Gopinath et al. [22], Jobstmann et al. [30], generate constraints and solve them for patches that are correct by construction (i.e., guaranteed to adhere to a specification or pass a test suite). In contrast, *generate-and-validate* repair approaches, e.g., GenProg [38], Pachika [17], PAR [33], Debroy and Wong [18], Prophet [41], find multiple repair candidates (e.g., using stochastic search or invariant inferences) and verify them against given specifications. *Learning-based* repair approaches, e.g., Fixminer [35], DLFix [40], DeepDelta [48], iFixR [36], learns fixes from repair examples.

Some program synthesis and repair researches, e.g., [5, 37, 47, 53, 54, 66, 67], integrate existing tools, e.g., test-input generation or symbolic execution, to synthesize desired programs. Such integrations are common in modern synthesis techniques, which generalize program repair by searching for code to fulfill given specifications, including the multi-disciplinary ExCAPE project [1] and the SyGuS competition [2], and have produced many practical and useful tools such as Sketch that generates low-level bit-stream programs [65], Autograder that provides feedback on programming homework [64], and FlashFill that constructs Excel macros [23, 24].

PMAXSAT has been utilized by many researches to locate bugs and generate patches for imperative programs. For example, AgxFaults [57] uses PMAXSAT to locate faults by finding the minimal set of suspicious statements, DirectFIx [46] applies PMAXSAT to find the simplest patch with minimal changes, TINTIN [14] uses PMAXSAT for regression aware localization and repair, and ConcBugAssist [31] uses PMAXSAT and model checking to diagnose concurrency bugs. These techniques focus on imperative programs and formulate statements and control flows into PMAXSAT formulas, which are not applicable in the context of a declarative language like Alloy. ATR is different from them in that it uses PMAXSAT to generate "tests" and uses the "tests" to prune the search space.

Alloy Repair. Comparing to APR and synthesis approaches for imperative programs, there are few works for APR techniques for Alloy. ARepair [74] is the first APR approach for Alloy and generates fixes for Alloy specification violating tests. This work

Table 3: Comparison to ARepair and BeAFix.

Benchmark			**ARepair**			**BeAFix**			**ATR**		
	Model	**# Cases**	**correct**	**overfit**	**time(s)**	**correct**	**overfit**	**time(s)**	**correct**	**overfit**	**time(s)**
Alloy4Fun	classroom	999	88	713	59.2	387	0	2221.7	**688**	0	601.8
	cv	138	2	116	2.2	**82**	0	1472	38	0	376.2
	graphs	283	19	243	1.4	232	0	673.1	**260**	0	156.9
	lts	249	1	238	0.4	41	0	3013	**70**	0	487.3
	production	61	27	32	2	**56**	0	311.4	43	0	332.8
	trash	206	48	140	4.7	183	0	405	**187**	0	368.3
	Summary	1936	185	1492	**31.5**	981	0	1788.3	**1286**	0	368.3
ARepair	addr	1	1	0	5.1	1	0	0.5	1	0	0.7
	arr	2	2	0	4.7	2	0	2.4	1	0	1.9
	balancedBST	3	1	1	268.6	1	**0**	2400.1	0	1	584.9
	bempl	1	0	0	3.3	0	0	3600	0	1	3.3
	cd	2	0	2	2.2	2	0	0.7	2	0	92.2
	ctree	1	**1**	0	3.9	0	0	3600	0	0	3600
	dll	4	0	3	20.2	**3**	0	902	2	0	1615.2
	farmer	1	0	1	32.7	0	0	3600	0	0	3600
	fsm	2	**2**	0	3.9	1	**0**	1800.2	1	1	3.9
	grade	1	0	1	101.7	0	0	3600	0	1	1.8
	other	1	0	0	2.9	**1**	0	3.1	0	1	2.9
	student	19	2	10	248.7	**13**	**0**	1185.6	7	3	443.6
	Summary	38	9	18	**152.2**	24	**0**	1351.2	14	8	829.2
	Total	1974	194	1500	**91.9**	1005	**0**	1569.8	**1300**	8	364.4

relies AUnit [69] tests, which are unit tests represented as Alloy predicates, AlloyFL [75] fault localizer, which also use unit tests to identify suspicious program statements, and mutation-based repairs (e.g., [38] which attempt to repair bugs by randomly mutate Alloy expressions. However, ARepair suffers from overfitting due to using tests as oracle, from which the generated repairs pass all tests but can not generalize to unseen tests.

BeAFIX [25] mitigates the overfitting problem by repairing Alloy based on assertions, which are more natural than tests in Alloy development and represent large sets of tests. BeAFix does not perform fault localization and instead relies on the user providing suspicious statements. BeAFix finds repairs by mutation, but unlike existing mutation-based approaches that limit search space by mutation operations, BeAFix exhaustively searches all possible candidates up to a certain bound. BeAFix also uses Alloy counterexamples, however mainly for checking variabilization feasibility.

FLACK [77] aims to find suspicious Alloy expressions responsible for failing assertions. Similar to ATR, FLACK analyzes counterexamples and PMAXSAT instances to identify relevant information. ATR adapts and extends this approach to generate repairs. In particular, it generates candidates repairs using templates and uses information from differences among counterexample and satisying instances to prune candidates.

6 CONCLUSION

We introduce a new automatic program repair technique for declarative specifications written in the Alloy language. Our insight is that we can generate Alloy expressions that fix an assertion violation by analyzing and comparing the counterexamples and satisfying instances from the Alloy Analyzer. We present ATR, a tool that implements these ideas to generate candidate repair expressions in specification. ATR uses a PMAXSAT solver to find satisfying instances similar to counterexamples generated by the Alloy Analyzer, analyzes satisfying instances and counterexamples to prune candidate repair templates. Preliminary results on a large set Alloy benchmarks show that ATR is effective in repairing complex Alloy bugs that existing Alloy APR techinques cannot.

ACKNOWLEDGMENT

We thank the anonymous reviewers for helpful comments. This work was supported in part by the National Science Foundation under grant numbers 1755890, 1618132, 2139845, 1948536, and 2107035; and PICT 2017-2622 and 2019-2050 from the Argentine National Agency of Scientific and Technological Promotion (ANPCyT).

REFERENCES

[1] 2020. ExCAPE project. https://excape.cis.upenn.edu/.
[2] 2020. SyGuS. https://sygus.org/.
[3] Devdatta Akhawe, Adam Barth, Peifung E. Lam, John C. Mitchell, and Dawn Song. 2010. Towards a Formal Foundation of Web Security. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, July 17-19, 2010.* 290–304.
[4] Mohannad Alhanahnah, Clay Stevens, and Hamid Bagheri. 2020. Scalable analysis of interaction threats in IoT systems. In *ISSTA'20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020.* ACM, 272–285.
[5] Paul Attie, Ali Cherri, Kinan Dak Al Bab, Mohamad Sakr, and Jad Saklawi. 2015. Model and program repair via sat solving. In *MEMOCODE.* IEEE, 148–157.
[6] Hamid Bagheri, Eunsuk Kang, Sam Malek, and Daniel Jackson. 2015. Detection of Design Flaws in the Android Permission Protocol Through Bounded Verification. In *FM 2015: Formal Methods.* Lecture Notes in Computer Science, Vol. 9109. 73–89.

[7] Hamid Bagheri, Eunsuk Kang, Sam Malek, and Daniel Jackson. 2018. A formal approach for detection of security flaws in the android permission system. *Formal Aspects of Computing* 30, 5 (Sept. 2018), 525–544.

[8] Hamid Bagheri, Eunsuk Kang, and Niloofar Mansoor. 2020. Synthesis of assurance cases for software certification. In *ICSE-NIER 2020: 42nd International Conference on Software Engineering, New Ideas and Emerging Results, Seoul, South Korea, 27 June - 19 July, 2020*. ACM, 61–64.

[9] Hamid Bagheri, Alireza Sadeghi, Reyhaneh Jabbarvand, and Sam Malek. 2016. Practical, Formal Synthesis and Automatic Enforcement of Security Policies for Android. In *Proceedings of DSN*. 514–525.

[10] Hamid Bagheri and Kevin Sullivan. 2012. Pol: Specification-Driven Synthesis of Architectural Code Frameworks for Platform-Based Applications. In *Proceedings of the 11th ACM International Conference on Generative Programming and Component Engineering (GPCE'12)*. ACM, Dresden, Germany, 93–102.

[11] Hamid Bagheri and Kevin Sullivan. 2016. Model-driven synthesis of formally precise, stylized software architectures. *Formal Aspects of Computing* 28, 3 (March 2016), 441–467.

[12] Hamid Bagheri, Chong Tang, and Kevin Sullivan. 2014. TradeMaker: Automated Dynamic Analysis of Synthesized Tradespaces. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. ACM, New York, NY, USA, 106–116.

[13] Hamid Bagheri, Jianghao Wang, Jarod Aerts, Negar Ghorbani, and Sam Malek. 2021. Flair: efficient analysis of Android inter-component vulnerabilities in response to incremental changes. *Empir. Softw. Eng.* 26, 3 (2021), 54.

[14] Rohan Bavishi, Awanish Pandey, and Subhajit Roy. 2016. To Be Precise: Regression Aware Debugging. 51, 10 (oct 2016), 897–915. https://doi.org/10.1145/3022671.2984014

[15] Roderick Bloem, Rolf Drechsler, Görschwin Fey, Alexander Finder, Georg Hofferek, Robert Könighofer, Jaan Raik, Urmas Repinski, and André Sülflow. 2013. FoREnSiC– An Automatic Debugging Environment for C Programs. In *Hardware and Software: Verification and Testing*. Springer Berlin Heidelberg, Berlin, Heidelberg, 260–265.

[16] Alcino Cunha, Nuno Macedo, and Tiago Guimarães. 2014. Target Oriented Relational Model Finding. In *Fundamental Approaches to Software Engineering*. Springer Berlin Heidelberg, Berlin, Heidelberg, 17–31.

[17] Valentin Dallmeier, Andreas Zeller, and Bertrand Meyer. 2009. Generating Fixes from Object Behavior Anomalies. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*. IEEE Computer Society, USA, 550–554.

[18] Vidroha Debroy and W Eric Wong. 2010. Using mutation to automatically suggest fixes for faulty programs. In *Software Testing, Verification and Validation*. IEEE, 65–74.

[19] Greg Dennis, Felix ShengHo Chang, and Daniel Jackson. 2006. Modular Verification of Code with SAT. In *Proc. of ISSTA*.

[20] Julian Dolby, Mandana Vaziri, and Frank Tip. 2007. Finding Bugs Efficiently with a SAT Solver. In *Proceedings of ESEC/FSE*. ACM Press, 195–204.

[21] Juan P. Galeotti, Nicolás Rosner, Carlos G. Lopez Pombo, and Marcelo F. Frias. 2013. TACO: Efficient SAT-Based Bounded Verification Using Symmetry Breaking and Tight Bounds. *IEEE Transactions on Software Engineering* 39, 9 (Sept. 2013), 1283–1307.

[22] Divya Gopinath, Muhammad Zubair Malik, and Sarfraz Khurshid. 2011. Specification-Based Program Repair Using SAT. In *Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011*, Vol. 6605. Springer, 173–188.

[23] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-output Examples. In *POPL* (Austin, Texas, USA). ACM, 317–330.

[24] Sumit Gulwani, William R. Harris, and Rishabh Singh. 2012. Spreadsheet Data Manipulation Using Examples. *Commun. ACM* 55, 8 (Aug. 2012), 97–105.

[25] Simón Gutiérrez Brida, Germán Regis, Guolong Zheng, Hamid Bagheri, ThanhVu Nguyen, Nazareno Aguirre, and Marcelo Frias. 2021. Bounded Exhaustive Search of Alloy Specification Repairs. In *Proceedings of the 43rd ACM/IEEE International Conference on Software Engineering ICSE 2021, Virtual, May 2021*.

[26] Daniel Jackson. 2002. Alloy: A Lightweight Object Modelling Notation. *ACM Trans. Softw. Eng. Methodol.* 11, 2 (April 2002), 256–290.

[27] Daniel Jackson. 2006. *Software Abstractions - Logic, Language, and Analysis*. MIT Press.

[28] Daniel Jackson, Ilya Shlyakhter, and Manu Sridharan. 2001. A Micromodularity Mechanism. In *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-9)*. ACM, New York, NY, USA, 62–73.

[29] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. 2011. Automated Atomicity-Violation Fixing. *SIGPLAN Not.* 46, 6 (June 2011), 389–400.

[30] Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. 2005. Program repair as a game. In *Computer Aided Verification*. 226–238.

[31] Sepideh Khoshnood, Markus Kusano, and Chao Wang. 2015. ConcBugAssist: Constraint Solving for Diagnosis and Repair of Concurrency Bugs *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 12 pages. https://doi.org/10.1145/2771783.2771798

[32] Sarfraz Khurshid and Darko Marinov. 2004. TestEra: Specification-based Testing of Java Programs Using SAT. *Automated Software Engineering* 11 (2004), 2004.

[33] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. IEEE Computer Society, 802–811.

[34] Jung Soo Kim and David Garlana. 2010. Analyzing architectural styles. *Journal of Systems and Software* 83, 7 (2010), 1216–1235.

[35] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. Fixminer: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering* (2020), 1–45.

[36] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. 2019. IFixR: Bug Report Driven Program Repair. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. New York, NY, USA, 314–325.

[37] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: Syntax- and Semantic-Guided Repair Synthesis via Programming by Examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 593–604.

[38] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Trans. Software Eng.* 38, 1 (2012), 54–72.

[39] Chu Min Li and Felip Manyà. 2009. MaxSAT, Hard and Soft Constraints. In *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications, Vol. 185. IOS Press, 613–631.

[40] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. DLFix: Context-Based Code Transformation Learning for Automated Program Repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 602–614.

[41] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 298–312.

[42] Nuno Macedo, Alcino Cunha, José Pereira, Renato Carvalho, Ricardo Silva, Ana C. R. Paiva, Miguel Sozinho Ramalho, and Daniel Castro Silva. 2020. Experiences on Teaching Alloy with an Automated Assessment Platform. In *Rigorous State-Based Methods - 7th International Conference, ABZ 2020, Ulm, Germany, May 27-29, 2020, Proceedings*, Vol. 12071. Springer, 61–77.

[43] Ferney A. Maldonado-Lopez, Jaime Chavarriaga, and Yezid Donoso. 2014. Detecting Network Policy Conflicts Using Alloy. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8477)*. Springer, 314–317.

[44] Niloofar Mansoor, Jonathan A. Saddler, Bruno Vieira Resende e Silva, Hamid Bagheri, Myra B. Cohen, and Shane Farritor. 2018. Modeling and testing a family of surgical robots: an experience report. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. ACM, 785–790.

[45] Darko Marinov and Sarfraz Khurshid. 2001. Testera: A novel framework for automated testing of Java programs. In *Proceedings of the IEEE International Conference on Automated Software Engineering (ASE'01)*.

[46] Sergey Mechtaev, J. Yi, and A. Roychoudhury. 2015. DirectFix: Looking for Simple Program Repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 448–458. https://doi.org/10.1109/ICSE.2015.63

[47] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 691–701. https://doi.org/10.1145/2884781.2884807

[48] Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. 2019. DeepDelta: Learning to Repair Compilation Errors *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 925–936.

[49] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. 2016. Reducing Combinatorics in GUI Testing of Android Applications. In *Proceedings ICSE*. 559–570.

[50] Joseph P. Near and Daniel Jackson. 2014. Derailer: Interactive Security Analysis for Web Applications. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. ACM, New York, NY, USA, 587–598.

[51] Joseph P. Near and Daniel Jackson. 2016. Finding security bugs in web applications using a catalog of access control patterns. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 947–958.

[52] Timothy Nelson, Christopher Barratt, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. 2010. The Margrave Tool for Firewall Analysis. In *Uncovering the Secrets of System Administration: Proceedings of the 24th Large Installation System Administration Conference, LISA 2010, San Jose, CA, USA, November 7-12, 2010*. USENIX Association.

[53] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*. 772–781.

[54] ThanhVu Nguyen, Westley Weimer, Deepak Kapur, and Stephanie Forrest. 2017. Connecting program synthesis and reachability: Automatic program repair using test-input generation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 301–318.

[55] Jaideep Nijjar and Tevfik Bultan. 2011. Bounded Verification of Ruby on Rails Data Models. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (Toronto, Ontario, Canada) *(ISSTA '11)*. ACM, New York, NY, USA, 67–77.

[56] Suhas Pai, Yash Sharma, Sunil Kumar, Radhika M. Pai, and Sanjay Singh. 2011. Formal Verification of OAuth 2.0 Using Alloy Framework. In *Communication Systems and Network Technologies (CSNT), 2011 International Conference on*. 655–659.

[57] Quang-Ngoc Phung and Eunseok Lee. 2021. Incremental Formula-Based Fix Localization. *Applied Sciences* 11, 1 (2021). https://doi.org/10.3390/app11010303

[58] Pablo Ponzio, Nazareno Aguirre, Marcelo F. Frias, and Willem Visser. 2016. Field-exhaustive Testing. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 908–919.

[59] Mark Clifford Reynolds. 2012. *Security Analysis of Bytecode Interpreters Using Alloy*. PhD Thesis. Boston University, Boston, MA, USA. Advisor(s) Kfoury, Assaf J.

[60] N. Rosner, Juan Galeotti, Santiago Bermudez, Guido Marucci Blas, Santiago Perez De Rosso, Lucas Pizzagalli, Luciano Zemin, and Marcelo F. Frias. 2013. Parallel Bounded Analysis in Code with Rich Invariants by Refinement of Field Bounds. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. 23–33.

[61] Natali Ruchansky and Davide Proserpio. 2013. A (not) NICE way to verify the openflow switch specification: formal modelling of the openflow switch using alloy. In *ACM SIGCOMM 2013 Conference, SIGCOMM 2013, Hong Kong, August 12-16, 2013*. ACM, 527–528.

[62] Bradley R. Schmerl, Jeff Gennari, Alireza Sadeghi, Hamid Bagheri, Sam Malek, Javier Cámara, and David Garlan. 2016. Architecture Modeling and Analysis of Security in Android Systems. In *Software Architecture - 10th European Conference, ECSA 2016, Copenhagen, Denmark, November 28 - December 2, 2016, Proceedings*. 274–290.

[63] Danhua Shao, Sarfraz Khurshid, and Dewayne Perry. 2007. Whispec: White-box testing of libraries using declarative specifications. In *Proc. of LCSD'07*.

[64] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. In *PLDI*. ACM, 15–26.

[65] Armando Solar-Lezama. 2013. Program sketching. *Int. J. Softw. Tools Technol. Transf.* 15, 5-6 (2013), 475–495.

[66] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. 2010. From program verification to program synthesis. In *POPL*. ACM, 313–326.

[67] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S Foster. 2013. Template-based program verification and program synthesis. *Soft. Tools for Technol. Transfer* 15, 5-6 (2013), 497–518.

[68] Clay Stevens and Hamid Bagheri. 2020. Reducing run-time adaptation space via analysis of possible utility bounds. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*. ACM, 1522–1534.

[69] Allison Sullivan, Kaiyuan Wang, and Sarfraz Khurshid. 2018. AUnit: A Test Automation Tool for Alloy. In *11th IEEE International Conference on Software Testing, Verification and Validation, ICST 2018, Västerås, Sweden, April 9-13, 2018*. IEEE Computer Society, 398–403.

[70] Mana Taghdiri. 2004. Inferring Specifications to Detect Errors in Code. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE '04)*. IEEE Computer Society, Washington, DC, USA, 144–153.

[71] Mana Taghdiri. 2007. *Automating Modular Program Verification by Refining Specifications*. PhD Thesis. Massachusetts Institute of Technology.

[72] Emina Torlak, Mana Taghdiri, Greg Dennis, and Joseph P. Near. 2013. Applications and extensions of Alloy: past, present and future. *Mathematical Structures in Computer Science* 23, 4 (2013), 915–933.

[73] Kaiyuan Wang, Allison Sullivan, and Sarfraz Khurshid. 2018. Automated model repair for Alloy. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. ACM, 577–588.

[74] Kaiyuan Wang, Allison Sullivan, and Sarfraz Khurshid. 2019. ARepair: a repair framework for alloy. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. IEEE / ACM, 103–106.

[75] Kaiyuan Wang, Allison Sullivan, Darko Marinov, and Sarfraz Khurshid. 2020. Fault Localization for Declarative Models in Alloy. In *31st IEEE International Symposium on Software Reliability Engineering, ISSRE 2020, Coimbra, Portugal, October 12-15, 2020*. IEEE, 391–402.

[76] Guolong Zheng, Quang Loc Le, ThanhVu Nguyen, and Quoc-Sang Phan. 2019. Automatic Data Structure Repair Using Separation Logic. *SIGSOFT Softw. Eng. Notes* 43, 4 (Jan. 2019), 66.

[77] Guolong Zheng, ThanhVu Nguyen, Simón Gutiérrez-Brida, Germán Regis, Marcelo Frias, Nazareno Aguirre, and Hamid Bagheri. 2021. FLACK: Counterexample-Guided Fault Localization for Alloy Models. In *Proceedings of the 43rd ACM/IEEE International Conference on Software Engineering ICSE 2021, Virtual (originally Madrid, Spain), 23-29 May 2021*.