ARepair: A Repair Framework for Alloy

Kaiyuan Wang
Google Inc, USA
kaiyuanw@google.com

Allison Sullivan
North Carolina A&T State University, USA
aksullivan@ncat.edu

Sarfraz Khurshid
University of Texas at Austin, USA
khurshid@utexas.edu

***Abstract*—Researchers have proposed many automated program repair techniques for imperative languages, e.g. Java. However, little work has been done to repair programs written in declarative languages, e.g. Alloy. We proposed ARepair, the first automated program repair technique for faulty Alloy models. ARepair takes as input a faulty Alloy model and a set of tests that capture the desired model properties, and produces a fixed model that passes all tests. ARepair uses tests written for the recently introduced AUnit framework, which provides a notion of unit testing for Alloy models. In this paper, we describes our Java implementation of ARepair, which is a command-line tool, released as an open-source project on GitHub. Our experimental results show that ARepair is able to fix 28 out of 38 real-world faulty models we collected. The demo video for ARepair can be found at https://youtu.be/436drvWvbEU.**

I. Introduction

Automated program repair (APR) techniques have been widely studied in the last decades. Researchers have proposed many APR techniques for imperative languages [16], [2], [6], e.g. Java. However, little work has been done to repair programs written in declarative languages, e.g. Alloy. We proposed ARepair [10], the first APR technique for Alloy. We choose Alloy because it is used in a variety of domains [4], [5], [17]. Alloy comes with a SAT-based analyzer [9] that performs analysis in a bounded scope on the universe of discourse. The analyzer has an evaluator that is able to evaluate the concrete value of a given Alloy expression or formula.

Recently, a number of Alloy extensions have been developed. AUnit [7] defines the notion of unit testing in Alloy by introducing AUnit tests, which capture the desired behaviors of Alloy models. MuAlloy [8], [11] defines mutation operators for Alloy grammar and performs mutation testing on Alloy models. MuAlloy is also able to generate AUnit tests that kill all non-equivalent first order mutant models. RexGen [12] designs various ways to generate semantically non-equivalent relational expressions in Alloy. ASketch [15], [13] is able to synthesize missing fragments of a partial Alloy model such that all the given AUnit tests pass. AlloyFL [14] designs various ways to locate faults in an Alloy model with failing AUnit tests. ARepair [10] is built on top of these previous techniques. Specifically, ARepair uses (1) AUnit tests automatically generated by MuAlloy to capture the desired model properties; (2) AlloyFL to locate faults based on the failing AUnit tests; (3) ASketch to create partial Alloy models with holes based on certain heuristics; (4) RexGen to generate candidate expressions for completing the holes; (5) the Alloy evaluator to validate the generated patches against tests.

This papers describes our Java implementation of ARepair, which is a command-line tool that we have released as an open-source project (https://github.com/kaiyuanw/ARepair). ARepair follows the standard *generate-and-validate* (G&V) approach [3], which takes as input a faulty Alloy model that triggers some failing tests and an AUnit test suite. ARepair then searches through the candidate space and tries to fix the model such that all tests pass. To fix the model, ARepair may run for multiple iterations. At each iteration, ARepair applies a change to the model such that some failing tests become passing while passing tests stay passing. In the end, this greedy approach either finds a patch that makes all tests passing or fails to fix the model. We evaluate ARepair and show that it is able to fix 28 out of 38 real faulty models.

II. Example and Background

A. Java Class Diagram

Figure 1 shows an faulty Alloy model that was written by a graduate student. The model is intended to express properties of a Java class diagram. We show part of the model that are relevant to the faults. Lines 1-2 declare the basic types in the problem: a notion of "`Class`" (line 1: "`sig`" denotes a set and introduces a type) and an "`Object`" class (line 2: "`one`" denotes a singleton set and "`extends`" denotes a subset relation). Each class may extend ("`ext`") at most one super class and there is a single Object class in the Java class hierarchy. The predicate "`ObjectNoExt`" in lines 3-6 should state that the Object class does not extend any class (as described in the comment at line 5). The student incorrectly states that for every class "`c`", the Object class is not a super class of "`c`" following the class hierarchy (line 6). The predicate "`AllExtObject`" in lines 7-10 should state that every class other than the Object class itself is a subclass of the Object class (as described in the comment at line 9). The student incorrectly states that for every class "`c`" which are not the Object class, "`c`" is one of its super classes including "`c`" itself. We do not show other parts of the model because the rest of the model is correct.

B. AUnit Test

An AUnit test is a pair of a model valuation and a run command. The valuation can be encoded as an Alloy test predicate and the run command can have an "`expect`" keyword which indicates the expected satisfiability of the test predicate. ARepair uses AUnit tests to locate faults in Alloy models and validate the correctness of candidate patches. Figure 2 shows two failing AUnit tests that capture some properties that

```
 1. sig Class { ext: lone Class }
 2. one sig Object extends Class {}
 3. pred ObjectNoExt() {
 4.   // Object does not extend any class.
 5.   // Correct: no Object.ext
 6.   all c: Class | Object !in c.^ext }
 7. pred AllExtObject() {
 8.   // Each class expect Object is a sub-class of Object.
 9.   // Correct: all c: Class - Object | c in Object.^~ext
10.   all c: Class - Object | c in c.*ext } ...
```

Fig. 1: Java Class Diagram

```
pred test1 {
  some disj Obj0: Object { some disj Obj0, Cls0: Class {
    Object = Obj0 and Class = Obj0 + Cls0
    ext = Obj0->Cls0 + Cls0->Cls0 and ObjectNoExt[] } } }
run test1 for 3 expect 0
pred test2 {
  some disj Obj0: Object { some disj Obj0, Cls0, Cls1: Class {
    Object = Obj0 and Class = Obj0 + Cls0 + Cls1
    ext = Cls0->Cls1 + Cls1->Cls0 and AllExtObject[] } } }
run test2 for 3 expect 0
```

Fig. 2: Failing AUnit Tests

should not appear in the correct model. The "`test1`" predicate states that the "`Object`" set has a single atom "`Object0`" and the "`Class`" set consists of "`Object0`" and an atom "`Class0`". "`Object0`" extends "`Class0`" and "`Class0`" extends itself. Since the Object class has a super class, we expect "`test1`" to be unsatisfiable when invoking "`ObjectNoExt`". The "`test2`" predicate states that "`Object`" set has a single atom "`Object0`"; the "`Class`" set consists of "`Object0`", and two atoms "`Class0`" and "`Class1`". "`Class0`" extends "`Class1`" and "`Class1`" extends "`Class0`". Since the Object class is not a super class of "`Class0`" and "`Class1`", we expect "`test2`" to be unsatisfiable when invoking "`AllExtObject`". Both tests are actually satisfiable and thus failed due to the faults in Figure 1.

C. *AlloyFL*

AlloyFL is a mutation-based fault localization technique that performs mutations on the Alloy AST node granularity [14]. AlloyFL is composed of a mutation engine, a equivalence checker, a suspiciousness calculator and a AST node ranker. The mutation engine supports a variety of mutation operators. For example, unary operator insertion (*UOI*) inserts an unary operator before expressions, e.g. "`a.b`" to "`a.~b`". Binary operator replacement (*BOR*) replaces binary operators, e.g. "`a=>b`" to "`a<=>b`". The equivalence checker [11] is able to detect if the mutated model is semantically equivalent to the original model within a given scope. The suspiciousness calculator runs all the tests against the semantically non-equivalent mutated model and computes a suspiciousness score based on a heuristic formula, e.g. Ochiai [1]. Each mutable AST node is assigned a suspiciousness score and the maximum score is used in case multiple mutation operators apply to the same AST node. The AST node ranker then ranks all the AST nodes in descending order of their suspiciousness. Then, the final ranked list of suspicious nodes are modified by ARepair to fix the bug.

III. Technique

We briefly discuss the ARepair technique because of lack of space. More details can be found at our technical paper [10].

Figure 3 shows different components of ARepair and how these components are connected. ARepair takes as input a faulty Alloy model and a set of AUnit tests. The inputs are fed into AlloyFL, which returns a ranked list of suspicious AST nodes as described in Section C. Sometimes, the mutation of a suspicious node is a potential fix, in which case, ARepair applies the mutation of the most suspicious node if the mutation makes some failing tests pass while preserving passing test results. Otherwise, for each suspicious node, ARepair creates holes at each level of the AST in a bottom-up fashion. Suppose a suspicious AST has D depth, then ARepair first replaces all nodes of depth D with holes and tries to synthesize code fragments for those holes such that some failing tests pass. If no such patch is found, then ARepair replaces all nodes of depth $D-1$ with holes and repeats the same process. If no patch is found for holes at depth 0, then ARepair tries the next suspicious AST and repeats the entire process.

To make the synthesis more efficient, the structure analyzer statically finds the type of each subnode that is replaced with a hole, so that later we can generate a set of candidate fragments with the same type to fill in the hole. This is crucial because replacing a hole with a code fragment whose type is different from the original code may result in a compilation warning. For instance, if we want to replace "`c.*ext`" of type "`Class`" in Figure 1 line 10 with integer "`1`" of type "`Int`", then the formula "`c !in 1`" is trivially true and the analyzer raises an warning stating that the left operand and right operand of "`!in`" are always disjoint. The structure analyzer also finds the variables in scope of each hole, and passes the variables and the hole type to RexGen for generating candidate code fragments.

RexGen is able to generate a set of Alloy expressions bounded by a given size [12]. It takes as input the basic variables used to compose more complex expressions, a given target type and a pre-defined bound, and produces as output a set of non-equivalent Alloy expressions following the grammar shown in [10]. The size of an expression is defined as the number of descendant nodes in the AST representation of the expression. ARepair sets the size of the generated expressions for the holes in the maximum depth to S and increases the size up to $S+3$ as the depth of the holes decreases, where S can be chosen by the user. The generated expressions are cached by the variables in scope that are used to create the leaves and the bounded expression size, so that RexGen does not need to regenerated expressions for another hole if it shares the same variables and the bounded expression size.

The set of non-equivalent expressions returned by RexGen is fed into the synthesizer, which fills holes with the generated expressions. Since there might be multiple holes at the same level and the number of candidate expressions for each hole could be over thousands, the total size of the search space is huge. ARepair implements two search strategies, i.e. the *all-combination* search strategy (AC for short) and the *base-choice* search strategy (BC for short). The AC strategy partitions the candidate expressions of each hole into k parts, i.e. from P_1 to P_k. Expressions in P_i have smaller or equal sizes compared to expressions in P_{i+1}. The strategy creates a tuple of partitions <P_i^1, P_j^2, ...> by taking the Cartesian product of partitions

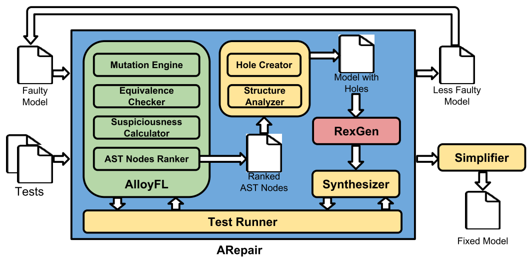

Fig. 3: ARepair Component Diagram

across all holes, where P_x^y denotes the x^{th} partition of hole y. Tuples with partitions of smaller x are ranked first. Then the strategy takes the Cartesian product of candidate expressions across partitions in each tuple, and searches through such combinations of expressions first. This strategy prioritizes combinations of candidate expressions with smaller sizes. Since the search space for AC strategy is still huge, we set a bound on the number of combinations to try when searching each level of holes. As soon as a combination of expressions that makes some failing tests pass while preserving passing test results is found, the strategy stop searching and applies the change to the model. The BC strategy fixes the candidate expressions of all holes except one and tries all expressions in that hole to find the expression that makes the most number of failing tests pass while preserving the passing test results. The strategy then replaces the hole with the found expression and performs the search on the next hole until all holes are exhausted. Both strategies are greedy (to reduce the exploration space) but they work relatively well in practice.

The test runner helps the synthesizer to validate candidate expressions against the test suite mostly using evaluator calls instead of the expensive constraint solving. ARepair builds a dependency graph for each test predicate. Each test depends on all facts, including explicit facts and implicit signature constraints, and the predicates and functions the test transitively invokes. Initially, ARepair removes all constraints in the model and invokes a constraint solver to obtain the instance represented in each test predicate. For example, to get an instance represented by `"test1"` in Figure 2, ARepair removes (1) the signature multiplicity constraint (`"one"` in Figure 1 line 2); (2) all explicitly declared Alloy predicates (e.g. `"ObjectNoExt"` and `"AllExtObject"`) and their invocations (e.g. `"ObjectNoExt[]"` in `"test1"`). Then, ARepair obtains the representing Alloy instance of `"test1"` by invoking `"run test1"` command. The satisfiability of a test is determined by the satisfiability of each fact and the body of the test. If any fact or the body of the test is unsatisfiable under the instance, then the test is unsatisfiable. Otherwise, it is satisfiable. During the search, we know which Alloy paragraph (i.e. signatures, facts, predicates, functions and assertions) is changed and thus can infer the affected Alloy paragraphs, including the affected test predicates. ARepair evaluates the affected test body and the dependent affected facts to determine test satisfiability. This process does not involve expensive constraint solving and it reduces the number of evaluator calls.

The evaluator-based approach to determine test satisfiability can be optimized by hierarchical caching. The idea is to reuse the previously evaluated result of a formula if its subformulas evaluate to the same set of values as some subformulas evaluated before. For example, if ARepair creates a hole to replace `"c.*ext"` in Figure 1 line 10, i.e. `"all c: Class - Object | c in ??"`. To evaluate the satisfiability of `"AllExtObject"`, suppose the first expression the synthesizer tries is `"none"`, then ARepair builds a hierarchical cache as follows. Since `"none"` evaluates to $\emptyset$, ARepair creates a mapping <`"none"`,$\emptyset$> for hole `"??"`. Then, ARepair evaluates the entire body of `"AllExtObject"`, which evaluates to false because `"Class0"` does not belong to the empty set, and creates a mapping <`"pred AllExtObject() {...c in ∅}"`,false> for the predicate `"AllExtObject"`. Note that the key is the string representation of `"AllExtObject"` with all descendant holes replaced by their valuations under the instance of `"test2"`. Next, ARepair creates a mapping <`"false"`,false> where the key is the boolean result of `"AllExtObject"` as a string and the value is the boolean result of `"test2"` by evaluating the body of `"test2"`. If the next candidate expression of hole `"??"` is `"c-Class"`, which evaluates to $\emptyset$, then we immediately know that `"test2"` evaluates to the same result as when hole `"??"` is `"none"`. Because the new keys ARepair computes based on `"c-Class"` all are all in the cache, we only need to invoke the evaluator once to evaluate `"c-Class"` instead of evaluating the body of `"test2"`. In general, the hierarchical cache reduces the input size but increases the number of evaluator calls. In practice, we observe speed-ups for a majority of the repair problems, but a few problems do suffer from a slow-down.

In summary, ARepair iteratively applies candidate patches from either AlloyFL's mutation or the synthesizer, and makes some failing tests pass while preserving the passing test results. After several iterations, if all tests pass, then ARepair finds a potential fix. Otherwise, the repair fails. The iterative approach allows ARepair to fix models with multiple faults or a single fault spanning multiple locations. If a potential fix is found, a simplifier makes the fixed model look more natural to the developer, e.g. simplifying `"c-none"` to `"c"`.

IV. Usage

In this section, we describe how users can invoke ARepair. More details can be found on the ARepair GitHub homepage.

To repair a faulty Alloy model, run `"./arepair.sh --run -m <arg> -t <arg> -s <arg> -c <arg> -g <arg> [-e] [-h <arg>] [-p <arg>] [-d <arg>]"` or `"./arepair.sh --run --model-path <arg> --test-path <arg> --scope <arg> --minimum-cost <arg> --search-strategy <arg> [--enable-cache] [--max-try-per-hole <arg>] [--partition-num <arg>] [--max-try-per-depth <arg>]"`.

- `"-m,--model-path"`: This argument is *required*. Pass the faulty Alloy model to repair as the argument.
- `"-t,--test-path"`: This argument is *required*. Pass the AUnit test suite which capture the desired properties of the expected model as the argument.
- `"-s,--scope"`: This argument is *required*. Pass the Alloy scope for repairing the faulty Alloy model as the argument.

The scope should be larger than or equal to the minimum scope necessary to run all AUnit tests properly.

- "`-c,--minimum-cost`": This argument is *required*. Pass the minimum size of the expression to generate as the argument. RexGen will generate expressions of the specified size for the deepest level of the suspicious AST.
- "`-g,--search-strategy`": This argument is *required*. Pass the search strategy to use for the internal synthesizer as the argument. The value should be either "`all-combinations`" or "`base-choice`".
- "`-e,--enable-cache`": This argument is optional. If this argument is specified, ARepair uses the hierarchical caching for repair. Otherwise, it does not.
- "`-h,--max-try-per-hole`": This argument is optional and is used when the search strategy is "`base-choice`". Pass the maximum number of candidate expressions to consider for each hole during repair as the argument. If the argument is not specified, a default value of 1000 is used.
- "`-p,--partition-num`": This argument is optional and is used when the search strategy is "`all-combinations`". Pass the number of partitions of the search space for a given hole as the argument. If the argument is not specified, a default value of 10 is used.
- "`-d,--max-try-per-depth`": This argument is optional and is used when the search strategy is "`all-combination`". Pass the maximum number of combinations of candidate expressions to consider for each level/depth of holes during repair as the argument. If the argument is not specified, a default value of 10000 is used.

For each run, for each iteration, the tool reports: (1) fault localization time; (2) the expression generation time; (3) the search space; (4) whether the current iteration successfully makes some failing tests pass but preserves the passing test results; (5) whether the fix comes from the mutation or the synthesizer; and (6) the model after the fix. Finally, the tool reports the simplified fixed model if all tests pass. Otherwise, the tool reports the latest state of the partially fixed model.

V. Evaluation

We evaluate ARepair on a machine running Ubuntu 16.04 LTS with 2.4GHz Intel Xeon CPU and 16 GB RAM. We collect 38 real faulty Alloy models with 62 individual faults. We use the default setting to run the experiment [10]. AC is able to fix 24 models and 31 faults. BC is able to fix 26 models and 42 faults. Additionally, AC times out ($\geq$ 15h) for 12 models while BC finishes all models in 15h. AC is able to fix 2 models that BC is not able to fix. BC is able to fix 5 models that AC is not able to fix. In total, ARepair is able to fix 28 models when considering both AC and BC.

To validate the correctness of the fixed model, we check the equivalence of the fixed model (with all tests passed) and the correct model using both manual inspection and alloy analyzer (under a bounded scope). We then inspect patches that are syntactically different from human-written patches and find that these patches are easy to understand in general. There are rare cases that ARepair generates some complex patches that can be further simplified through semantic reasoning.

```
 6.- all c: Class | Object !in c.^ext }
 6.+ all c: Class | Object !in c.^~ext }
...
10.- all c: Class - Object | c in c.*ext }
10.+ all c: Class - Object | c.*ext - Object != c.*ext }
```

Fig. 4: ARepair Patch for Class Diagram

We illustrate the repair process using the example in Figure 1. In the first iteration, ARepair finds that some mutation applied by AlloyFL can make some failing tests passing so it applies the BOR mutation which mutates "`c in c.*ext`" to "`c != c.*ext`". In the second iteration, ARepair find that another mutation applied by AlloyFL can make fewer tests failing, including "`test2`" in Figure 2, so it applies the UOI mutation which mutates "`Object !in c.^ext`" in line 6 Figure 1 to "`Object !in c.^~ext`". In the third iteration, ARepair is able to find a patch that makes all tests pass by replacing the left "`c`" in "`c != c.*ext`" with a synthesized expression "`c.*ext-Object`". The final patch is semantically equivalent to the correct patch written by the TA as shown in Figure 4.

VI. Conclusion

This paper introduces ARepair, an open-source, command-line tool for repairing faulty Alloy models. Our experiment shows that ARepair is able to fix 28 out of 38 real-world faulty models we collected.

References

[1] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund, "A practical evaluation of spectrum-based fault localization," *JSS*, 2009.

[2] Fan Long and M. Rinard, "Automatic patch generation by learning correct code," in *POPL*, 2016.

[3] J. Hua, M. Zhang, K. Wang, and S. Khurshid, "Towards practical program repair with on-demand candidate generation," in *ICSE*, 2018.

[4] T. Nelson, C. Barratt, D. J. Dougherty, K. Fisler, and S. Krishnamurthi, "The margrave tool for firewall analysis," in *LISA*, 2010.

[5] N. Ruchansky and D. Proserpio, "A (not) nice way to verify the openflow switch specification: Formal modelling of the openflow switch using alloy," *SIGCOMM*, 2013.

[6] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "Elixir: effective object oriented program repair," in *ASE*, 2017.

[7] A. Sullivan, K. Wang, and S. Khurshid, "AUnit: A Test Automation Tool for Alloy," in *ICST*, 2018.

[8] A. Sullivan, K. Wang, R. N. Zaeem, and S. Khurshid, "Automated test generation and mutation testing for Alloy," in *ICST*, 2017.

[9] E. Torlak and D. Jackson, "Kodkod: A relational model finder," in *TACAS*, 2007.

[10] K. Wang, A. Sullivan, and S. Khurshid, "Automated model repair for alloy," in *ASE*, 2018.

[11] ——, "MuAlloy: A Mutation Testing Framework for Alloy," in *ICSE*, 2018.

[12] K. Wang, A. Sullivan, M. Koukoutos, D. Marinov, and S. Khurshid, "Systematic generation of non-equivalent expressions for relational algebra," in *ABZ*, 2018.

[13] K. Wang, A. Sullivan, D. Marinov, and S. Khurshid, "Asketch: A sketching framework for alloy," in *FSE*, 2018.

[14] ——, "Fault localization for declarative models in Alloy," in *eprint arXiv:1807.08707*, 2018.

[15] ——, "Solver-based sketching Alloy models using test valuations," in *ABZ*, 2018.

[16] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *ICSE*, 2009.

[17] P. Zave, "Using lightweight modeling to understand chord," *SIGCOMM Comput. Commun. Rev.*, pp. 49–57, 2012.