ICEBAR: Feedback-Driven Iterative Repair of Alloy Specifications

Simón Gutiérrez Brida
University of Rio Cuarto and CONICET
Argentina

Germán Regis
University of Rio Cuarto
Argentina

Guolong Zheng
University of Nebraska-Lincoln
USA

Hamid Bagheri
University of Nebraska-Lincoln
USA

ThanhVu Nguyen
George Mason University
USA

Nazareno Aguirre
University of Rio Cuarto and CONICET
Argentina

Marcelo Frias
Buenos Aires Institute of Technology and CONICET
Argentina

ABSTRACT

Automated program repair (APR) techniques have shown great success in automatically finding fixes for programs in programming languages such as C or Java. In this work, we focus on repairing *formal specifications*, in particular for the Alloy specification language. As opposed to most APR tools, our approach to repair Alloy specifications, named ICEBAR, does not use test-based oracles for patch assessment. Instead, ICEBAR relies on the use of property-based oracles, commonly found in Alloy specifications as predicates and assertions. These property-based oracles define stronger conditions for patch assessment, thus reducing the notorious overfitting issue caused by using test-based oracles, typically observed in APR contexts. Moreover, as assertions and predicates are inherent to Alloy, whereas test cases are not, our tool is potentially more appealing to Alloy users than test-based Alloy repair tools.

At a high level, ICEBAR is an iterative, counterexample-based process, that generates and validates repair candidates. ICEBAR receives a faulty Alloy specification with a failing property-based oracle, and uses Alloy's counterexamples to build tests and feed ARepair, a test-based Alloy repair tool, in order to produce a repair candidate. The candidate is then checked against the property oracle for overfitting: if the candidate passes, a repair has been found; if not, further counterexamples are generated to construct tests and enhance the test suite, and the process is iterated. ICEBAR includes different mechanisms, with different degrees of reliability, to generate counterexamples from failing predicates and assertions.

Our evaluation shows that ICEBAR significantly improves over ARepair, in both reducing overfitting and improving the repair rate. Moreover, ICEBAR shows that iterative refinement allows us to significantly improve a state-of-the-art tool for automated repair of Alloy specifications *without* any modifications to the tool.

CCS CONCEPTS

• **Software and its engineering → Formal software verification**; **Software testing and debugging**.

KEYWORDS

Formal Specification, Automated Repair, Alloy

ACM Reference Format:
Simón Gutiérrez Brida, Germán Regis, Guolong Zheng, Hamid Bagheri, ThanhVu Nguyen, Nazareno Aguirre, and Marcelo Frias. 2022. ICEBAR: Feedback-Driven Iterative Repair of Alloy Specifications. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3551349.3556944

1 INTRODUCTION

In the last decade or so, many automated program repair (APR) techniques have been introduced to fix non-trivial bugs [20, 26, 29, 30, 32, 34–36, 38, 39, 42, 45, 59, 60]. These techniques concentrate in *programs* (e.g., C and Java), and to maintain automation, they typically exploit common elements in program development. In particular, APR techniques typically require a test suite, e.g., for auxiliary tasks such as fault localization, but most importantly as an acceptance criterion for candidate patches [32]. That is, a fix candidate is usually considered a valid patch if it passes the test suite accompanying the program. This "test suites as oracles" situation makes the effectiveness of repair approaches strongly depend on the quality of the test suites. Moreover, the inherent partiality of tests as specifications make repair techniques subject to *overfitting*, i.e., the problem of producing patches that pass the corresponding test suite but do not fix the program in general [31, 44, 47, 53].

While repair techniques have emphasized the automated fixing of programs, other software artifacts such as *software specifications*

ASE '22, October 10–14, 2022, Rochester, MI, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3556944

are also subject to hard-to-repair defects, and thus can profit from automated repair. Software specifications are used for problem domain modeling and for software design. Unfortunately, correctly capturing an informal problem description and devising sound design ideas are challenging, error-prone, and arduous tasks. Of course, defects in specifications can lead to incorrect implementations, and thus detecting, localizing, and fixing those errors are highly relevant areas of research. For these reasons, borrowing approaches for program repair and applying these for software specification repair is worthwhile. This presents, simultaneously, difficulties for the application of repair techniques, and opportunities to improve the effectiveness of these techniques. For instance, in the context of formal specifications, it is usual to have property-based specification "oracles", in the form of properties that are expected to hold from the specification assumptions. Oracles in the form of test cases, that are commonly found for programs, are more rarely seen accompanying specifications. Notice how the absence of tests prevents the direct application of automated repair approaches, and at the same time the existence of stronger property-based oracles would provide an opportunity for reducing the overfitting of program repair techniques, in the context of specifications.

Based on the above observations, we present ICEBAR (Iterative CounterExample Based Alloy Repair), a technique and tool for automatically repairing formal specifications written in the Alloy language [21]. Alloy is an expressive formal specification language with support for SAT-solving based property checking, that has many applications in software development, including telecommunication protocol design [61], security analysis in mobile applications [6, 8, 9], automated test generation [5, 24, 41], and bounded program verification [14–16]. ICEBAR builds upon ARepair [55], an automated repair tool for Alloy specifications that, in the spirit of traditional APR, requires test cases, both for fault localization and for patch acceptance checking. ARepair introduces the difficulty of getting test cases for the specification to be repaired, which are typically *not* part of Alloy specifications. It is also seriously affected by overfitting, and does not exploit property-based oracles likely to be present in specifications. ICEBAR, on the other hand, receives an Alloy specification with a failing property-based oracle, and uses Alloy's counterexamples to automatically build tests and feed ARepair, to search for a repair candidate. The produced candidate is then checked against the property oracle for overfitting: if the candidate satisfies the properties, a repair has been found; if not, further counterexamples are generated to build tests that enhance the test suite, disregarding the currently produced candidate, and the process is iterated. Thus, our obtained repairs cannot be overfitting with respect to the property-based oracles (which are in turn stronger than the tests), but may still be overfitting in the sense that they may not conform with the developer's intention, beyond what the property-based oracles express.

Alloy property checking comes, essentially, in two forms: asserting that a property is a consequence of the specification and querying for the satisfiability of a property, in conjunction with the specification. Thus, two kinds of failing properties can be found: an expected property does not hold, and thus the solver produces a counterexample, and a property expected to be consistent with the specification is found inconsistent. In the former, a test case can be produced from the counterexample, that we can reliably incorporate into a test suite to run ARepair. In the latter, we may produce model instances from which to build test cases by *relaxing* the property or specification (with the sole purpose of producing instances), but the produced test cases are unreliable, in the sense that one does not know a priori if these represent behaviors that the correct specification should allow for, or not. ICEBAR includes mechanisms to produce both kinds of test cases, and processes these differently according to their reliability, to automatically guide the search for specification repairs.

Besides presenting our technique in detail, we perform an experimental evaluation over two Alloy repair benchmarks (consisting of faulty Alloy specifications used in evaluating previous Alloy repair techniques), that show that the technique effectively improves ARepair's ability to repair Alloy specifications. More precisely, ICEBAR improves ARepair's repair rate by $5.7X$ and $2.3X$, respectively, in the two benchmarks. Our approach is also able to repair specifications that are beyond what other tools for Alloy repair [11] can handle. The tool and all the experimental data associated with its evaluation are available as a replication package [3]. We also provide a Github repository with ICEBAR's implementation [4].

2 BACKGROUND AND MOTIVATION

We now introduce Alloy modeling, and discuss specification validation, debugging and repair, as a motivation for our technique.

2.1 Alloy Modeling

Consider the problem of formally capturing *linked lists*, and the notions involved in this data representation, such as the structural description of the linear object organization, the constraints that enforce (a)cyclicity, etc. A language that may be used to formally specify linked lists is Alloy [21]. Alloy is a formal specification language with numerous applications in the modeling and analysis of software. The language has been designed taking into account specification *readability*, *expressiveness* and *analyzability* (among other design dimensions). Regarding readability, specifications in Alloy involve a few abstractions, with precise semantics, that are easy to grasp for software developers. The style of specifications in Alloy is *model oriented*, organized around the definition of data domains, and relations between these domains. While the language is *formal* and *relational* in nature, its specifications retain an intuitive reading, and Alloy model descriptions resemble object-oriented modeling. With respect to expressiveness, specification constraints are expressed in Alloy's relational logic [21], essentially a first-order logic complemented with relational operators (in particular reflexive-transitive and transitive closures), that extend the expressiveness of the language beyond that of predicate logic. Finally, regarding analyzability, Alloy supports fully automated analysis, by reducing bounded satisfiability and validity checking of Alloy specifications to SAT solving. Alloy modeling is supported by the Alloy Analyzer, the tool that allows one to write models and automatically analyze them via the above mentioned SAT-based procedure.

Returning to the problem of specifying linked lists, a formal attempt to capturing this concept in the Alloy language is shown in Figure 1. This model (this is the way in which specifications are referred to in the context of Alloy) is in fact taken verbatim from the case studies used as part of the evaluation in [55] (the model is called student5). The model mentions four data domains, which in Alloy are captured via signatures (and formally introduce unary relations, i.e., sets). Signatures List and Node define domains for lists and nodes, respectively. The predefined Int signature (integers) is used to represent elements stored in list's nodes. Signatures can declare typed *fields*, which give structure to the specification, and formally declare relations from the signature where they are defined, to the corresponding type. For instance, the two fields of signature Node, namely link and elem, declare binary relations from the Node set (the domain associated with this signature) to Node and Int, respectively. The cardinality constraints in these fields simply indicate that each node may be mapped to any number of nodes and elements via the corresponding relations (set indicates "zero or more"). Finally, signature Boolean declares a domain with two atoms (constants) True and False, using abstract signatures (sets with no proper elements), signature extension (relational inclusion, which can model inheritance), and cardinality constraints on signatures (a "one" signature has exactly one element, and different signatures extending a given one must be disjoint).

Alloy specifications can also include *facts*, that represent assumptions of the specification, and are captured using Alloy's relational logic. Fact CardinalityConstraints in the specification constrains further the cardinalities of fields: every list has at most one header (operator lone indicates the expression can have cardinality zero or one); every node can have at most one link; and every node has *exactly* one element (operator one indicates the expression must have cardinality one).

Alloy specifications typically involve the definition of *predicates*, parameterized formulas that can be used to describe properties, model operations, characterize families of scenarios, and even be called from other predicates and facts. For instance, predicate Loop describes a property of list structures: a list This satisfies Loop iff: *(i)* all nodes are reachable from the header of This through traversals of link (all is universal quantification, dot denotes relational composition, * is reflexive-transitive closure, and in denotes relational inclusion); and *(ii)* either This has no header, or there exists exactly one node reachable from This.header through link, that can reach itself through link (quantifier one is "exists exactly one", and ^ denotes transitive closure).

Predicates can also be used to model operations. Predicate Contains is an example of this usage, where some predicate parameters represent inputs (This and x) and some the outputs (result); this predicate captures the *contains* operation, that checks whether an element belong to a list or not.

Finally, Alloy models can include *assertions* to represent *intended properties* of the specification, i.e., properties that should hold in every scenario where the specification assumptions are satisfied. Our sample model has no assertion; for the sake of presentation, we will introduce one. An expected property of the model may be that, under the assumption of the facts, if RepOk holds and Contains returns True for some element, then the list cannot be empty:

```
assert ContainsTrueImpliesNonEmptyList {
    all l: List | RepOk[l] and (some x: Int | Contains[l, x, True])
                  implies some l.header
}
```

```
sig List {
    header: set Node
}

sig Node {
    link: set Node,
    elem: set Int
}

// Correct
fact CardinalityConstraints {
    all l: List | lone l.header
    all n: Node | lone n.link
    all n: Node | one n.elem
}

// Correct
pred Loop(This: List) {
    all n: Node | n in This.header.*link
    no This.header || one n: This.header.*link | n in n.^link
}

// Underconstraint.  Should consider link = n1 -> n2 without loop.
pred Sorted(This: List) {
    all n: This.header.*link | n.elem <= n.link.elem
}

pred RepOk(This: List) {
    Loop[This]
    Sorted[This]
}

// Correct
pred Count(This: List, x: Int, result: Int) {
    RepOk[This]
    result = #{n:This.header.*link | n.elem = x}
}

abstract sig Boolean {}
one sig True, False extends Boolean {}

// Correct
pred Contains (This: List, x: Int, result: Boolean) {
    RepOk[This]
    #{n: This.header.*link | x in n.elem} != 0 => result = True
    #{n: This.header.*link | x in n.elem} = 0 => result = False
}

fact IGNORE {
  one List
  List.header.*link = Node
}
```

Figure 1: A faulty Alloy model of linked lists.

Alloy models can be automatically analyzed in essentially two ways, that are reduced to SAT solving. On one hand, given a predicate and a so-called *scope*, defining a maximum number of elements for each of the domains in the specification, Alloy Analyzer can check for the *satisfiability* (or alternatively, the unsatisfiability) of the predicate *within* the scope. This analysis basically answers the question: *does there exist an instance of the specification that does not exceed the scope, and satisfies the facts and the predicate?* Since the scope makes the number of potential instances finite, the satisfiability problem, for a given scope, becomes decidable. Alloy Analyzer implements this bounded satisfiability check by reducing it to a propositional satisfiability problem, that is in turn solved by an off-the-shelf SAT solver. Similarly, given an assertion, and a scope, Alloy Analyzer can check for its *validity*, within the scope. It then answers the question: *Does this assertion hold for all instances of* *the specification that do not exceed the scope and satisfy the facts?* Of course, this can be turned into a satisfiability problem simply negating the assertion, and Alloy Analyzer checks it that way, via a reduction to SAT solving. Two commands issuing these kinds of checks for our sample model are the following:

```
run RepOk for 5 but exactly 1 List expect 1
check ContainsTrueImpliesNonEmptyList for 10 expect 0
```

Together with the scope, these commands indicate an expectation (a boolean represented with 1 for true and 0 for false), since Alloy Analyzer allows one to indicate if a predicate is expected to be satisfiable/unsatisfiable, and an assertion is expected to be valid or invalid. Note that when, as a result of analyzing a command, an analysis check is satisfiable (e.g., a predicate is found satisfiable, or an assertion is found invalid), an instance witnessing this result is produced. The developer can use it as an example of a satisfying predicate, or as a violation of an intended assertion, to help confirm his or her modeling decisions, and improve or debug the model, when the result contradicts the expectation.

2.2 Alloy Test Cases

As we just described, specification instances or scenarios are present during Alloy modeling, as a result of the analysis. Instances witnessing the satisfiability of a predicate, or counterexamples to the validity of an assertion, are generated and returned to the developer, as a result of performing automated SAT-based analysis. But scenarios are typically not described within the same Alloy model, in fact, Alloy instances are not even part of Alloy model's syntax. However, since the language is sufficiently expressive, a developer can in fact specify concrete scenarios into an Alloy model. This is used in many practical settings, in particular to make the constraint solver "fill in" a *partial* scenario [54].

Concrete specification scenarios can also be useful as a vehicle for *validating* formal specifications. The idea here is that, via the definition of various concrete specification instances, and appropriately indicating if these correspond to behaviors that are expected or not from the specification, one would also be able to explicitly state expectations on the specification. These will be scenario-specific, as with unit tests in the context of programs. Since assessing a concrete scenario, and deciding whether it represents a desirable behavior or not, is easier than interpreting a formula, the addition of concrete scenarios, appropriately tagged as "desirable" or "not desirable", into the Alloy specification setting has the value of more directly capturing some expected properties of the specification, allowing developers to gain confidence in their specifications. This observation has motivated the introduction of AUnit [51], a language and tool that helps developers to write *Alloy test cases*. AUnit exploits the expressive power of Alloy to provide a syntax for instance-based properties, that is reducible to Alloy itself.

As a simple example, in our linked list specification, we would expect the empty list to satisfy RepOk. This expectation can be captured using AUnit:

```
val EmptyList {
  no List.header
  @cmd{ RepOk() }
}
@Test ValidEmptyList: run EmptyList
```

The `val` new section corresponds to the introduction of a parameterless Alloy predicate [51]. These represent *valuations* in the logical sense (valuations of formulas). Notice how a valuation can include a *command*, the formula evaluated in the defined instance (or family of instances) and that should evaluate to true to consider that the test passes. AUnit allows one to define and run these test cases, it defines notions of coverage and mutation for Alloy, which the tool can measure, and even provides mechanisms to automatically generate tests based on coverage and mutation [52].

2.3 Specification Defects and Repair

Even in Alloy, where the language has been designed to give specifications a clear intuitive reading, it is common to make mistakes while attempting to capture a specification, often due to overlooked restrictions, or wrongly imposing too restrictive constraints (making a specification stronger, in a logical sense, than necessary), as well as using the language incorrectly (e.g., misinterpreting the actual meaning of operators). Thus, specifications must be validated and debugged as is the case with source code. In Alloy, predicates capturing properties of a specification, as well as assertions, are the typical instruments for a developer to validate his or her specifications. The typical approach to validate specifications in Alloy involves both automated tasks and manual ones. First, developers use the Alloy Analyzer to check for the bounded validity of assertions, as well as for the satisfiability of certain predicates. If one of these checks fails, it indicates an analysis outcome contrary to the expectations, and thus the presence of specification defects or bugs. Moreover, developers can improve this scenario by considering scenario-specific checks on predicates and assertions, via the use of AUnit test cases as explained above.

In the above cases, we have an explicit specification *oracle*, given in terms of predicates and assertions, unit test cases, or both. If any of these fails during analysis, the corresponding specification can be deemed incorrect, and the debugging and specification repair phases are initiated. These latter steps are typically performed manually, and are arduous and time consuming. Thus, having automated support to aid developers in specification debugging and repair is important. Fortunately, some techniques for automated Alloy specification repair have been recently proposed. ARepair [55] was the first such technique; it borrows concepts from program repair, and applies them to automatically repair Alloy specifications, provided the specification has a *test suite* repair oracle. ARepair relies on Alloy test cases for fault localization (using a spectrum-based technique [58]), for guiding the repair process, and for deciding whether a patch has been found (patch acceptance criterion) [55]. The technique uses mutation and systematic expression generation [57] for producing patch candidates, and is able to efficiently generate repairs that involve complex syntactic changes in the faulty specification. An alternative approach is BeAFix [11]. BeAFix does not necessarily require test cases: it can be applied with specification oracles based on standard Alloy predicates and assertions, as well as test cases, or their combination. BeAFix uses these oracles mainly as a patch acceptance criterion. It performs a bounded-exhaustive exploration of a space of mutation-based patches. Thus, it either finds a fix or guarantees that no such fix exists within the exploration bound. It also implies that the repairs that BeAFix finds are syntactically close to the faulty specification, leading to simpler patches, but at the same time preventing the technique from fixing specifications that require more involved syntactic changes.

These repair techniques are complementary, and have advantages and disadvantages. On one hand, being based on test cases, ARepair can be greatly affected by overfitting, or put it in another way, the quality of the patches greatly depends on the quality of the test cases. Moreover, even when property-based oracles are present in the specification, ARepair cannot profit from these, as it only considers test-based oracles. BeAFix, on the other hand, being bounded-exhaustive in nature, can suffer from scalability issues. More precisely, when the required patches are relatively distant (syntactically speaking) from the original faulty expression, it is unlikely that BeAFix would be able to find such patches, as it explores mutation-based candidate patches in breadth-first, and these grow exponentially with the depth of the search. As a concrete example, consider again our linked list specification. As indicated in the specification, it has a defect, in the definition of predicate `Sorted`. In order to run ARepair, one needs to provide test cases. Using AUnit, these test cases can be automatically generated (the instances are automatically generated, but of course these still have to be manually classified as "desirable" or "undesirable"). From such automatically generated tests, ARepair is able to find a patch, but an overfitting one: it passes the tests but it does not fix the issue. It is only after the ARepair developers added some manually crafted tests, that the tool fixes the specification. BeAFix, on the other hand, does not need tests and can use the specification assertions, but cannot produce a fix for this specification. Although the expression needed to repair it is in principle reproducible by BeAFix, the mutation depth required to reach it, and the search space traversal policy of the tool, make it infeasible for the tool to find this patch.

Our motivation is improving Alloy specification repair, by taking advantage of ARepair's efficiency and ability to generate complex repairs, and at the same time dealing with overfitting in a better way. Our approach works as follows. Given a faulty specification with property-based oracles and test cases, first we attempt to repair the specification using ARepair and the test cases. If a fix is found, we contrast it against the property based oracles for confirmation. If the fix passes the oracle, we are done. If on the other hand this fix is spurious, we use the Alloy Analyzer to produce a *new* test, which can be added to the suite disregarding the spurious fix. This process is iterated. In this way, the test generation is driven by the repair approach, and we reduce ARepair's overfitting by taking advantage of property based oracles.

3 THE ICEBAR APPROACH

3.1 Overview of the Technique

Test-based specification repair as realized in tools like ARepair offers a number of advantages. On one hand, it ports successful techniques applied in the context of program repair, to the context of specifications. Approaches to fault localization, fix candidate construction and incremental fix candidate checking, are all well executed by ARepair, taking inspiration from similar techniques for programs. Moreover, being based on a non-exhaustive approach for patch generation, it constitutes a more efficient alternative to repair, especially in contrast with BeAFix. However, as all techniques based on tests, ARepair suffers from overfitting to a greater extent, compared to techniques using stronger repair oracles. As usual in the context of program repair, the test suite employed for fault localization and repair is *static*, i.e., it does not change during the repair process. This mechanism is graphically summarized as part of Figure 2 (highlighted in green), where is clear what the inputs to ARepair are, and how ARepair either fails to repair a specification, or produces a patch that makes the input test suite pass. Notice how ARepair views the faulty Alloy specification as a whole, without discerning parts of it that actually represent property-based oracles, such as predicates and assertions.

Our observation here is that, in the context of Alloy, it is typical to have property-based oracles as part of the Alloy specification, that may impose more general constraints on the expectations of the specification, and that may be exploited to aid ARepair. Figure 2 shows an overview of our technique. It starts from a given faulty specification, with a failing property-based oracle, and a (possibly empty) test suite. It first runs ARepair with the test suite to attempt to produce a patch. If a patch is found, we know that is passes the test suite; we take this candidate and contrast it against the property-based oracle in the specification. If, again, the specification meets this oracle, we consider the patch a proper fix. If, on the other hand, some property fails in the candidate patch, we use Alloy Analyzer to produce instances, which are in turn translated into *new* unit tests, to complement the original test suite and start over the ARepair process. If a fix that passes both the corresponding test suite and property-based oracle is eventually found, it is returned to the user.

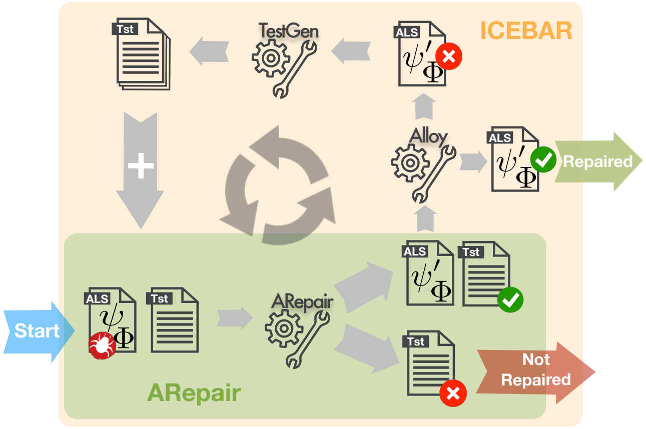

Figure 2: An overview of ICEBAR

The time factor is also important in the above described scenario. Since ARepair can rather efficiently produce a fix (or an indication that no fix is possible), it is possible to iterate the process after validating a fix against a property-based oracle and strengthening the test suite based on the resulting violations, while still being efficient.

Our general description of the technique involves three main stages: *model repair*, *model validation*, and *test generation*. We describe below how these stages are implemented. We also discuss certain technical challenges, in relation to how instances are produced from violated properties in spurious patches, and whether these can effectively guarantee the progress in the repair.

3.2 Model Repair

The model repair phase of our approach consists of a black-box usage of the ARepair tool. Our ICEBAR approach makes iterations on the execution of ARepair, each of which is fully independent, i.e., no information other than the current faulty specification and test suite is passed between subsequent runs of ARepair.

ICEBAR is parametric on the Alloy model repair technique. It may use any repair technique that expects as inputs a specification and a test suite for it. The current version of ICEBAR uses ARepair. It is important to remark though that, since the technique performs multiple iterative executions of the repair approach, its efficiency strongly depends on that of the tool being iterated. More precisely, while a variant of the technique may be devised for non-test based repair tools like BeAFix, the cost of running each instance (iteration) of this tool would make our approach inapplicable. Notice that our technique collects the property-based oracles from the faulty specification itself (these are defined as part of the specification). As these will be used as a stronger check on candidate fixes, they are assumed correct and are not subject to *direct* modifications during the repair process. For instance, assume that the specification includes an assertion, that we consider an oracle, stating the following:

$$\forall x \cdot p(x) \rightarrow q(x)$$

where p and q are predicates in the specification. This formula is assumed correct, and no syntactic modification is allowed on it. However, predicates p and q, being part of the specification, may be deemed incorrect by ARepair, and thus trigger modifications, indirectly changing the semantics of the assertion. This is perfectly tolerated by our approach.

3.3 Model Validation

Alloy models are validated by writing properties stating expectations on the specification, which are analyzed via *commands*. We call these *property-based oracles*. The two traditional ones are predicate satisfiability (stating a property that the developer expects to be satisfiable with the model assumptions) and assertion checking (checking that a given assertion is a consequence of the model assumptions). As described earlier, Alloy models can also be validated against specific scenarios, in particular using AUnit test cases. We call the latter *test-based oracles*. Our assumption regarding model validation is that the to-be-repaired specifications have failing property-based oracles. Property-based oracles are inherent to Alloy specifications, and thus this assumption is easy to satisfy.

Notice that the use of predicates in Alloy specifications exceeds the statement of intended properties, as these also serve the purpose of organizing a specification in terms of auxiliary formulas, as well as for describing model operations, etc. We may ask the developer to manually distinguish the specification from the oracle, as proposed in [11]. Or, as we proceed in this paper, consider the property-based oracle to be the set of all predicates and assertions which have corresponding analysis commands. Without loss of generality, we assume that predicate commands always have satisfiability expectations, and assertion commands have validity expectations (note that in Alloy, one can also state that expects an assertion to fail, or a predicate to be unsatisfiable).

3.4 Test generation

The aim of the test generation phase is to produce new tests, after a model repair has produced an "incorrect" patch, i.e., a patch that passes the current test suite, but does not satisfy the model's property-based oracle. These new tests will be used so that a new incremental execution of model repair can be triggered. When validating an Alloy model against a property-based oracle, there are two possible situations indicating the model is incorrect: having an instance that satisfies the model assumptions but violates an expected assertion (a *counterexample*), and not having instances of a predicate expected to be consistent with the model assumptions (an *unsatisfiable* outcome in the Alloy analysis). Our test generation phase must be able to produce new test cases, to iterate the specification repair process, in both situations. Moreover, the generated test cases should strive to guarantee that, once incorporated into the test suite for repair, the previously produced incorrect patch cannot be generated as a repair candidate.

These two forms of property-based oracle violation are clearly asymmetric, as one produces an instance (a counterexample) that can be straightforwardly turned into a test case, while the other gives an unsatisfiability outcome, from which generating a test case is less direct. We will describe below how tests are specifically generated, in each of these cases.

3.4.1 *Dealing with Violated Assertions.*

Let us discuss violations to assertions first. The situation is as follows. We have an Alloy model M, a property-based oracle that includes an assertion A for M, and a test suite T, that fails on M. Model repair has been executed using M and T, producing a repair candidate M'. Although all tests in T pass in M' (since this model is the result of the repair model phase), M' does not satisfy the property-based oracle A. We have therefore obtained an instance I of M', where all facts $F_{M'}$ of M' hold, but assertion A fails.

The above-described erroneous situation captured by instance I can be due to either having a bug in the model's facts (e.g., these being too weak and thus allowing some instances that should not be allowed), or having a bug in the assertion A (e.g., the assertion captures a property stronger than intended). That is, intuitively either the facts are too weak, or the assertion is too strong (or both). We would like to turn I into a new test imposing the requirement that this erroneous situation should not be accepted in a patch of the specification. We therefore build a new test case t capturing the following property:

$$I \wedge (\neg F_{M'} \vee A)$$

That is, instance I will be either removed from feasible situations in a patched version of the model (i.e., it will violate the model facts), or it will satisfy the assertion. It is worth remarking here that $F_{M'}$ and A are not necessarily rigid formulas (otherwise the above will necessarily be always false): since both facts and assertions may call predicates and refer to functions in the model, changes in these predicates and functions during the repair process will change the semantics of the facts and assertion, turning the above test case into a "passing" test case. With the just created new test case t, we can go back to the repair phase, and attempt to repair M again, now with test suite $T' = T \cup \{t\}$.

Notice that the test cases we just described are *reliable* test cases: if we trust the assertion from which the test case was generated (notice that we only need to trust the assertion itself, but not necessarily the predicates or facts indirectly involved in it), we are certain that the generated counterexample described an undesired behavior, and thus we state so in the construction of the test case.

Finally, let us now argue why the new test case t will guarantee that we can reattempt to repair model M, and the previously produced spurious patch M' cannot be produced again. First, recall that M did not pass the test suite T; since T' contains T, we are sure that M does not pass T' and therefore the repair phase is enabled. Second, let M'' be a patch generated from M and test suite T'. This patch must necessarily pass all tests in T', in particular t; thus it cannot be M', since we know that t fails in M'.

3.4.2 Dealing with (Un)satisfiable Predicates. Let us now discuss failing predicates. The situation is as follows. We have an Alloy model M, a property-based oracle that includes a predicate P for M, and a test suite T, that fails on M. Model repair has been executed using M and T, producing a repair candidate M'. Again, although all tests in T pass in M', M' does not satisfy the property-based oracle P. We have therefore a situation in which P is *inconsistent* with the facts $F_{M'}$ of M', i.e., $F_{M'} \wedge P$ is unsatisfiable (within the employed analysis scope).

Since the unsatisfiability outcome does not produce an instance, an obvious first approach to attempt to produce a new test case is to find some alternative way of producing instances. First, if besides the failing predicate we have a failing assertion, then we are in the previously described situation: we can ignore the failing predicate, deal with the failing assertion and reliably go back to the repair phase. If, on the other hand, we have no failing assertion, we may still be able to produce instances by resorting to non-failing predicates. Assume P' is another part of the property-based oracle for M, and P' is non-failing, i.e., it is *consistent* with the facts $F_{M'}$. We can then produce an instance I satisfying $F_{M'} \wedge P'$, and generate a test case from I. However, although I satisfies P' (as expected), since I is an instance of the faulty specification M', we have no way of knowing whether I is a "desired" instance (a behavior of the intended specification), or not. Figure 3 graphically depicts the two possibilities for I: it is either an instance satisfying the predicate and the faulty specification which also satisfies the intended specification (green box), or is an instance satisfying the predicate and faulty specification *outside* the intended specification (red box). This leads to the following two alternative test cases:

$$\begin{aligned} t_{pos} &: \quad I \wedge F_{M'} \wedge P' \\ t_{neg} &: \quad I \wedge \neg(F_{M'} \wedge P') \end{aligned}$$

This is related to the oracle problem in software testing: in order to classify the actual obtained outputs for given inputs as correct or incorrect, one requires some reference. We do not have such a reference in this case[1]. Instead, we consider the two possibilities. We first hypothesize that the obtained satisfying instance is desirable (a "positive" instance), add t_{pos} as a test, and proceed with the search for a fix. If no fix is found, we backtrack, change the positive instance into a negative one (replace t_{pos} by t_{neg}), and continue from there. These test cases are then "unreliable", in the sense that we cannot be sure that they represent behaviors we want to include, or correspondingly exclude, in an eventual fix of the specification. In any case, when ICEBAR returns a fix, the set of all test cases considered for the corresponding repair (including the "unreliable" ones) are provided as feedback to the developer, who can further examine them as introduced hypotheses that led to the produced repair. If an inconsistency is found between these hypothesized scenarios and the intended behavior, the correction can be made, turning the corresponding test into a "reliable" one.

Finally, when we have no failing assertion nor non-failing predicate in the property-based oracle, we still need to produce instances from the unsatisfiability outcome for $F_{M'} \wedge P$. To obtain an instance in this case, we systematically relax $F_{M'}$ by incrementally removing conjuncts from this formula until a satisfiable formula is obtained or no more conjuncts are left to remove. Then, as with instances from non-failing predicates, and since the instances were obtained by relaxing the facts, it is reasonable to consider these instances as unreliable. We deal with these instances in the same way as with the instances generated from non-failing predicates.

It is worth noticing that unreliable tests may originate from non-failing predicates, and thus these tests may not be related with the defects in the specification. These can in fact lead to previously considered repair candidates. Nevertheless, these additional tests provide extra information that can guide ARepair towards fixes that, without these tests, would not be considered by the tool.

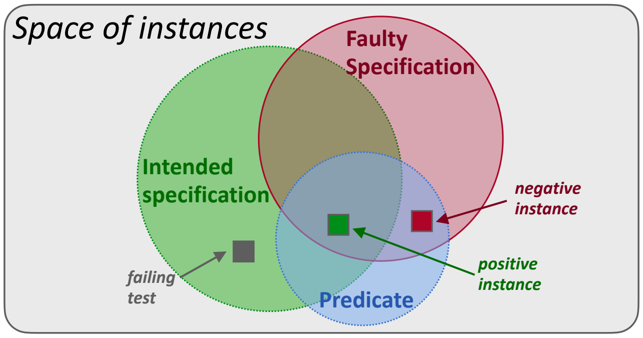

Figure 3: A description of how satisfying instances relate to the specification, the corresponding predicate, and the behavior intended to be captured by the specification.

4 EVALUATION

We present an experimental evaluation of ICEBAR. We consider three research questions.

RQ1 Does ICEBAR improve the repair effectiveness of ARepair?
RQ2 Are the tests produced by ICEBAR relevant to the repair process?
RQ3 How does ICEBAR compare to other Alloy repair tools?

To approach these research questions we consider two different benchmarks of faulty Alloy specifications: the Alloy4Fun project

[1] One may argue that the developer can serve as an oracle in this situation, and decide, for each generated test case, whether it represents a desired behavior or not. This alternative would seriously undermine automation, an important characteristic of our technique.

[37], and the benchmark of faulty specifications originally used to evaluate ARepair in [55]. Alloy4Fun consists of 6 different template models, with a total of 1936 human-written faulty variants, based on specific modeling assignments resolved by different students. The ARepair benchmark, on the other hand, is composed of 38 different faulty specifications drawn from various domains. For each of these models, we consider a corresponding correct version as oracle. All the experiments below were run on an AMD Ryzen Threadripper 64-Core Processor with 64GB RAM, under GNU/Linux.

Answering RQ1, i.e., evaluating whether ICEBAR can improve ARepair's repairability, aims at analyzing if our technique can replace the manual effort of designing test cases to improve repair effectiveness. We therefore only consider automatically produced test cases. More precisely, for the ARepair benchmark, whose test suites are composed of automatically generated tests (generated using the AUnit tool [51]) as well as manually produced tests, we only consider the automatically produced tests for ARepair (the objective of ICEBAR is to automatically produce the manual ones). In the case of the Alloy4Fun benchmark, where no tests are present, we run ARepair with test cases automatically generated using AUnit (using the corresponding correct specifications as oracles for these tests, as these needs to be manually classified as expected or unexpected). ICEBAR is run, for both benchmarks, starting with empty test suites (i.e., all tests are produced by ICEBAR's counterexample-driven approach). Each experiment was run with a 1 hour timeout. The results of this experiment are shown in Table 1. For each benchmark we report the total number of faulty specifications, the average number of tests that each tool was run with across all the corresponding benchmark's case studies, the number of correct fixes, overfitted repairs, cases for which the corresponding tools failed to produce a fix, and the average time it took each technique to process a case study in the corresponding benchmark. We also summarize the improvement in repairability, and for ICEBAR, report the average number of calls to ARepair (i.e., number of iterations of the counterexample-driven approach), and the average percentage of the repair time that was spent in executing ARepair (the remainder is the cost of generating tests, and validating repairs).

The objective of RQ2 is to determine whether there is substantial merit in ICEBAR's test generation approach, to improve ARepair's performance. More precisely, we want to analyze if the cases that ICEBAR improves over ARepair (i.e., when ARepair does not produce a fix or produces an overfitting fix, but ICEBAR correctly repairs the specification) are due to the tests that it produces, or are simply due to having more tests (or different tests) compared to ARepair. To answer RQ2, we performed the following experiment. For each case study, we consider the k number of tests that ICEBAR generated, both when repairing the corresponding specification, and when terminating due to timeout or without finding a repair. For the corresponding case study, we generated multiple random test suites of k test cases each, and ran ARepair with each random suite. Table 2 shows the results of these experiments, reporting the average across ARepair runs with random test suites. In relation to the different approaches that ICEBAR uses to produce new tests (see Section 3.4), it is worth remarking that, for the ARepair benchmark, 71% of the successful fixes used unreliable tests and, on average, 45% of the tests used in each of these cases were unreliable. Alloy4Fun cases required no use of unreliable tests.

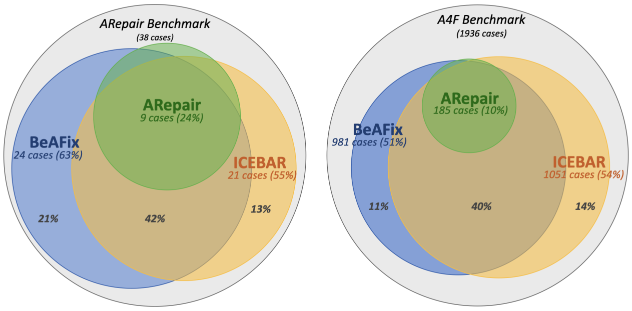

Figure 4: Overlappings in repaired cases

Let us now focus on RQ3. We compare the three tools we are aware of for automatically repairing Alloy specifications: ARepair, ICEBAR, and BeAFix [11]. The ARepair and ICEBAR runs are exactly as for RQ1, since we wish to evaluate these tools in a fully automated context. For BeAFix, we run each experiment with the property oracles, since as indicated in [11], this tool does not require test cases. Table 3 summarizes the results of these experiments. For each benchmark, tool and configuration, we report the number of correct fixes found, overfitted fixes and timeouts (TMO), and the average time for those cases in which the corresponding tool was able to produce a fix. Again, we use Alloy Analyzer to contrast fix candidates against a correct specification to check if a candidate is a correct fix. To further understand how these techniques complement each other, we depict in Figure 4, for each benchmark, which percentage of cases were only repaired by one of the tools but no the others, as well as the overlap between the techniques (percentage of cases that were correctly fixed by more than one tool). The fastest between ICEBAR and BeAFix is highlighted in boldface, together with its corresponding speedup over the other tool (notice that we do not consider speedup with respect to ARepair, which is the fastest but produces a high number of overfitting cases). We will further discuss these results later on.

4.1 Assessment

We first discuss the results of the experiments for RQ1. The experiments on the ARepair benchmark show that ICEBAR's tests improve upon ARepair's effectiveness, increasing the number of correct fixes from 9 to 21. Clearly, the complexity of the specifications and faults require higher number of ARepair calls by ICEBAR, with an average of around 38 total ARepair calls until a fix is found. Still, the 21 correct fixes found by ICEBAR are below the 26 correct fixes found by ARepair as reported in [55], although 17 out of the 26 required introducing manually designed tests (as opposed to ICEBAR, which fully automatically fixes the 21 cases).

We now analyze the experiments on the Alloy4Fun benchmark, consisting of 1936 faulty specifications. We see that more than 54% of these are correctly repaired by ICEBAR, with an average of about 6 ARepair calls until a fix is found. This is a significant improvement in repairability (or similarly, reduction in overfitting) compared to ARepair on automatically generated test suites, which can only repair about 10% of the cases in this benchmark. Regarding the experiments on this benchmark, it is worth observing in more

Table 1: Impact of ICEBAR in automatically adding scenarios to improve repairability

Benchmark		ARepair (automatic tests)				ICEBAR					Imp
Name	Total cases	AVG. # tests	# cases correct	overfit	*AVG. time (sec.)*	AVG. # tests	AVG. ARepair calls	#cases correct	*AVG. time (sec.)*	AVG. (%) ARepair time	
Alloy4Fun	**1936**	37,5	**185**	1492	*31,5*	9,8	5,9	**1051**	*915,1*	62%	**5,7X**
ARepair	**38**	42,9	**9**	18	*152,2*	18,6	38,2	**21**	*744,6*	56%	**2,3X**

Table 2: ICEBAR generated tests vs. Random generated tests

Benchmark			ICEBAR tests	Random tests
Name	Total cases	AVG. # tests	# cases correct	AVG. # cases correct
Alloy4Fun	1936	9,8	1051 (**54,2%**)	11 (**0,6%**)
ARepair	38	18,6	21 (**55,2%**)	0,7 (**1,8%**)

detail the cases that are not repaired by ARepair (nor correctly or overfitted). Out of the 359 cases that ARepair reports as not being able to repair, 92 (4.75% of the 1936 total number of cases) are due to grammar issues, i.e., Alloy operators present in the faulty specification that are not supported by ARepair. These are of course not repairable by ICEBAR either, since this tool is based on repeated executions of ARepair.

Regarding RQ2, ARepair can only repair, in average, about 1 case out of 38 for ARepair's benchmark when using random test suites (compared to the 21 that ICEBAR repairs). In the Alloy4Fun benchmark, ARepair can correctly fix 11 out of 1936 cases when using random test suites (compared to the 1051 cases that ICEBAR is able to repair from this benchmark). These results clearly show that the size of the test suites involved in ICEBAR's correct fixes is not the reason for the technique's effectiveness: "useful" tests, that actually guide the repair process, are necessary, and ICEBAR's counterexample-driven approach provides them.

Moving to RQ3, when observing the ARepair benchmark results, BeAFix and ICEBAR have similar fix rates, with BeAFix having a margin over ICEBAR. Neither of the tools completely subsumes the other in terms of repairability: 5 cases are repaired by ICEBAR but not by BeAFix, and 8 cases are repaired by BeAFix but not ICEBAR (notice that BeAFix repairs 4 cases that are not within the 26 repaired by ARepair with manually designed tests).

The results on the much larger Alloy4Fun benchmark allow us to further explore these issues. Again, ICEBAR and BeAFix have good fixing rates, 51% of the total are repaired by BeAFix, and 54% of the total are repaired by ICEBAR. The overlap figure shows that, again, none of these tools fully subsumes the other: they overlap in 773 cases (40%), 208 (11%) are repaired by BeAFix but not ICEBAR, and 278 (14%) are repaired by ICEBAR but not BeAFix. Here we should notice that 54 cases (2.8%) cannot be handled by ICEBAR because they use Alloy operators that ARepair does not currently support. Finally, if we compare the tools in terms of efficiency, in both benchmarks and for most cases, ICEBAR improves the running time and provides a significant speedup over BeAFix.

In order to qualitatively assess the kind of repairs that ICEBAR is able to generate over BeAFix, we manually examined the specifications that were correctly repaired by ICEBAR, but for which BeAFix was unable to produce a fix. Our analysis indicated that there are two different reasons preventing BeAFix from finding patches in these cases. On one hand, some patches require mutations to the expressions that, according to [11], BeAFix does not currently support. As an example, consider the following faulty expression from a specification in the Alloy4Fun benchmark:

```
no ~adj.adj
```

The specification is faulty, and a manual fix for it is:

```
no (iden & adj)
```

BeAFix cannot produce this fix, as it does not support the kind of expression mutations to arrive to it from the faulty specification. Notice that this may be overcome extending the set of mutation operators for the tool, which at the same time increases the search space, a critical issue for bounded-exhaustive approaches as the one BeAFix implements. For the above case, ICEBAR is able to find the following equivalent fix:

```
no (~(~(iden & adj))).adj
```

which is also beyond what BeAFix can produce.

Other cases can in principle be repaired by BeAFix, but would require multiple mutations to single faulty points. Since BeAFix explores the space of candidate repairs in breadth-first, i.e., from the syntactically closer to the faulty expression to more distant ones, the patches would require BeAFix to reach search depths that are infeasible for the tool. Consider for instance the following faulty expression from a specification in the Alloy4Fun benchmark:

```
all u:User , w:Work, i:Institution |
      w in u.profile && (w.source = u || w.source = i)
```

is repaired by ICEBAR with the following correct patch:

```
all u:User, w:Work, i:Source { ((w in (Source.(~source))) &&
     (((u + (profile.w)) = u) || ((w.source) = ((w.source) - u))))}}
```

It is clear that the syntactic distance between this patch and the original expression makes it infeasible to reach, within reasonable time, for BeAFix. The manual fix for this faulty specification is:

```
all u:User, w:Work | w in u.profile implies
          (u in w.source or some i:Institution | i in w.source
```

which cannot be produced by BeAFix either, since it requires expression mutations that the tool does not support.

All these experiments can be replicated as indicated in [3]. We also provide a Github repository with ICEBAR's implementation [4].

4.2 Threats to Validity

To account for threats to validity, we have considered various issues and made some design decisions in our experimental evaluation.

Table 3: Comparison of Alloy specification repair tools

Benchmark			**ARepair (automatic tests)**				**ICEBAR**				**BeAFix**			
	Model	Total	# cases			*AVG.*	# cases			*AVG.*	# cases			*AVG.*
		cases	correct	overfit	TMO	*time (s)*	correct	overfit	TMO	*time (s)*	correct	overfit	TMO	*time (s)*
Alloy4Fun	graphs	283	19	243	0	*1,4*	237	0	0	**_15,6_(43X)**	232	0	51	*673,1*
	lts	249	1	248	0	*0,4*	73	0	2	**_71,6_(42X)**	41	0	208	*3013,0*
	cv	138	2	116	0	*2,2*	86	0	7	**_479,4_(3X)**	82	0	56	*1472,0*
	production	61	27	32	0	*2*	36	0	0	**_71,7_(4X)**	56	0	5	*311,4*
	trash	206	48	140	0	*4,7*	195	0	2	**_101,5_(4X)**	183	0	23	*405,0*
	classroom	999	88	713	0	*59,2*	424	0	352	**_1661,5_(1X)**	387	0	612	*2221,7*
Summary		**1936**	**185**	**1492**	**0**	*31,5*	**1051**	**0**	**363**	***915,1***	**981**	**0**	**955**	*1788,3*
ARepair	addr	1	1	0	0	*5,1*	1	0	0	*9,1*	1	0	0	**_0,5_(18X)**
	arr	2	2	0	0	*4,7*	2	0	0	*47*	2	0	0	**_2,4_(19X)**
	balancedBST	3	1	1	0	*268,6*	2	0	1	**_1528,8_(1X)**	1	0	2	*2400,1*
	bempl	1	0	0	0	*3,3*	1	0	0	**_28,8_(124X)**	0	0	1	*3600,0*
	cd	2	0	2	0	*2,2*	2	0	0	*5,8*	2	0	0	**_0,7_(8X)**
	ctree	1	1	0	0	*3,9*	0	0	0	**_8,6_(418X)**	0	0	1	*3600,0*
	dll	4	0	3	0	*20,2*	3	0	1	*1742,9*	3	0	1	**_902_(2X)**
	farmer	1	0	1	0	*32,7*	0	0	0	*3600,0*	0	0	1	*3600,0*
	fsm	2	2	0	0	*3,9*	2	0	0	**_178,8_(10X)**	1	0	1	*1800,2*
	grade	1	0	1	0	*101,7*	1	0	0	**_4,5_(800X)**	0	0	1	*3600,0*
	other	1	0	0	0	*2,9*	0	0	0	*31,1*	1	0	0	**_3,1_(10X)**
	student	19	2	10	0	*248,7*	7	0	3	**_654,6_(2X)**	13	0	6	*1185,6*
Summary		**38**	**9**	**18**	**0**	*152,2*	**21**	**0**	**5**	***744,6***	**24**	**0**	**14**	*1351,2*

First, the subjects chosen for the experimental evaluation constitute a chief threat to external validity, in particular due to a lack of evidence that the results on these benchmarks will generalize to arbitrary specifications. We chose the considered benchmarks because they have been previously used to evaluate other Alloy repair techniques. Also, each specification has a correct version against which we can automatically contrast to assess overfitting, and all faulty specifications in these benchmarks are real human-written defects (as opposed to synthetic defects, for which generalizability is even harder to argue for). Note that we have considered the *whole* benchmarks, composed of a large number of faulty specifications, to avoid unintentional cherry-picking of specific subsets that may inadvertently favor or disfavor some of the tools. The benchmarks are admittedly composed of relatively small specifications, and they may not capture many aspects of complexity present in larger cases. They do, however, cover the entirety of the Alloy language, and involve some complex kernels characteristic of larger realistic Alloy specifications. We aim to promote comparative evaluation and reproducibility of experimental results, which requires using standard benchmarks and releasing implementations. The selected benchmarks are the ones that the Alloy specification repair considers; and as indicated above, ICEBAR and all the experimental data are available for experimental replication and validation.

ICEBAR depends on various external tools: Alloy Analyzer (as the core for instance generation and specification checking), and ARepair (for iteratively generating specification repairs from faulty specifications and test suites). While these tools are not proved correct, they have been used relatively extensively in the context of Alloy specification, and are considered robust, providing us a good degree of confidence on their soundness. We have taken official releases of these tools, and used them without any modifications (with the exception of a patch to a bug in ARepair that caused the tool to crash[2]), to avoid accidentally affecting these tools (we have inherited grammar limitations in ARepair, that we decided not to resolve, to use externally developed tools and not affect our experiments). Also, all these tools use SAT solvers as underlying constraint engines, which can have some degree of nondeterminism. We have run our experiments multiple times, and observed a negligible effect of low level SAT solving nondeterminism in these tools' results.

A main threat to internal validity is the verification of overfitting and correct patches. While these tasks may be performed manually, doing so would increase the error proneness, and would be ineffective for the large Alloy4Fun benchmark. Having correct versions of the to-be-repaired specifications allowed us to reduce this threat, by using Alloy Analyzer to perform the checking.

5 RELATED WORK

Alloy Repair. ICEBAR uses ARepair [55, 56] as an backend tool. ARepair repairs faulty Alloy specifications based on a set of Alloy *tests*. It combines imperative program repair techniques, like *sketching* [48] and mutation-based repairs [32]. ARepair can fix Alloy specifications with multiple bugs at different locations. It uses AlloyFL [58] to localize faults, and in the experimental evaluations, the test suites it requires to perform the repair have been automatically produced using AUnit [51], from a correct version of the faulty specification, to account for automated classification of generates tests as desired or undesired. By using a non-exhaustive approach for patch generation, ARepair effectively reduces its search space and the number of repair candidates. However, ARepair is more susceptible to produce overfitting repairs, due to its use of tests as oracles.

[2] We will submit the patch to the official repository of the tool, as a pull request.

ICEBAR mitigates overfitting by using property-based oracles, which represent large families of intended behavior tests. The aforementioned BeAFix tool [11] also repairs Alloy specifications using property based oracles. Contrary to ARepair, BeAFix does not perform fault localization, and instead can use arbitrary external fault localization tools, like AlloyFL [25, 58] or FLACK [63]. It also uses Alloy counterexamples, but instead of doing so only to reduce overfitting (as we do with ICEBAR), they are also exploited to prune certain partial fixes that can never lead to repairs. BeAFix finds repairs by mutations. Unlike most mutation based approaches, such as Gopinath et al. [17], Kim et al. [26] and Le Goues et al. [32], which restrict the search space by limiting mutation operations, BeAFix exhaustively searches for all possible candidates up to a certain bound. ICEBAR is designed for a test-based repair tool.

Automatic Program Repair and Program Synthesis. Due to the pressing demand for reliable software, automatic program repair has steadily gained research interests and produced many novel repair techniques. *Constraint-based* repair approaches, e.g., AFix [22], Angelix [39], SemFix [42], FoRenSiC [10], StarFix [62], Gopinath et al. [17], Jobstmann et al. [23], 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 [32], Pachika [12], PAR [26], Debroy and Wong [13], Prophet [36], 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 [27], DLFix [33], DeepDelta [40], iFixR [28], learns fixes from repair examples.

Some program synthesis and repair researches, e.g., [7, 30, 39, 42, 43, 49, 50], integrate existing tools, e.g., test-input generation or symbolic execution, to synthesize desired programs. In general, such integrations are common in modern synthesis works 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 [48], Autograder that provides feedback on programming homework [46], and FlashFill that constructs Excel macros [18, 19]. ICEBAR inherits these ideas and integrates a counterexample-driven repair process with ARepair. Although our technique is in essence similar to Sketch [48], the differences in context make our approach vary from sketching, in particular on how counterexamples are exploited. More precisely, the counterexamples that guide Sketch are *always* reliable, i.e., Sketch does not need to deal with the "weak oracle" problem inherent to our context, which leads us to assume that our produced instances are valid/invalid, and results in branching in the search for repairs.

6 CONCLUSION

We introduced ICEBAR, a technique that builds upon ARepair and deals with the inherent overfitting arising from using test suites as repair oracles. By using property-based oracles to validate fixes and strengthen the test suite when a violation is found, ICEBAR is effective in mitigating overfitting and generating complex patches. Furthermore, using properties as oracles (rather than tests) makes ICEBAR methodologically better suited for Alloy users, contributing to the Alloy model development process. ICEBAR is complementary to other state-of-the-art Alloy repair tools such as BeAFix: both tools produce repairs that the other cannot; BeAFix leads to shorter/clearer repairs, while ICEBAR is significantly more efficient.

This paper opens various lines for further work. Firstly, our technique is mainly focused on dealing with *overfitting*, the most pressing concern in automated repair. The readability of ICEBAR's patches is inherited from ARepair's, and clearly calls for improvement. Improving patch readability may be achieved either by improving ARepair, or by post-processing ICEBAR's output, e.g., applying some syntactic simplification techniques. We plan to explore both options as future work. Secondly, the effectiveness of ICEBAR may be affected by the "initial" test suite from which the repair is launched. Although we have not studied this issue in this paper, we plan to evaluate different strategies to generate initial test suites, and how these impact ICEBAR's repairability metrics.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful comments. This work was supported in part by awards CCF-1755890, CCF-1618132, CCF-2139845, and CCF-2124116 from the National Science Foundation; by Argentina's National Agency of Scientific and Technological Promotion (AN-PCyT) through projects PICT 2016-1384, 2017-1979 and 2017-2622; and by an Amazon Research Award.

REFERENCES

[1] 2020. ExCAPE project. https://excape.cis.upenn.edu/.

[2] 2020. SyGuS. https://sygus.org/.

[3] 2022. ICEBAR replication package. https://sites.google.com/view/icebar-evaluation.

[4] 2022 (accessed August 31, 2022). ICEBAR's Github Repository. https://github.com/saiema/ICEBAR.

[5] Pablo Abad, Nazareno Aguirre, Valeria S. Bengolea, Daniel Ciolek, Marcelo F. Frias, Juan P. Galeotti, Tom Maibaum, Mariano M. Moscato, Nicolás Rosner, and Ignacio Vissani. 2013. Improving Test Generation under Rich Contracts by Tight Bounds and Incremental SAT Solving. In *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*. IEEE Computer Society, 21–30. https://doi.org/10.1109/ICST.2013.46

[6] 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*, Sarfraz Khurshid and Corina S. Pasareanu (Eds.). ACM, 272–285. https://doi.org/10.1145/3395363.3397347

[7] 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.

[8] Hamid Bagheri, Eunsuk Kang, Sam Malek, and Daniel Jackson. 2018. A formal approach for detection of security flaws in the android permission system. *Formal Asp. Comput.* 30, 5 (2018), 525–544. https://doi.org/10.1007/s00165-017-0445-z

[9] Hamid Bagheri, Alireza Sadeghi, Reyhaneh Jabbarvand Behrouz, and Sam Malek. 2016. Practical, Formal Synthesis and Automatic Enforcement of Security Policies for Android. In *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016, Toulouse, France, June 28 - July 1, 2016*. IEEE Computer Society, 514–525. https://doi.org/10.1109/DSN.2016.53

[10] 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*, Armin Biere, Amir Nahir, and Tanja Vos (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 260–265.

[11] Simón Gutiérrez Brida, Germán Regis, Guolong Zheng, Hamid Bagheri, ThanhVu Nguyen, Nazareno Aguirre, and Marcelo F. Frias. 2021. Bounded Exhaustive Search of Alloy Specification Repairs. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 1135–1147. https://doi.org/10.1109/ICSE43902.2021.00105

[12] 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. https://doi.org/10.1109/ASE.2009.15

[13] 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.

[14] Greg Dennis, Felix Sheng-Ho Chang, and Daniel Jackson. 2006. Modular verification of code with SAT. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006*, Lori L. Pollock and Mauro Pezzè (Eds.). ACM, 109–120. https://doi.org/10.1145/1146238.1146251

[15] Juan P. Galeotti, Nicolás Rosner, Carlos Gustavo López Pombo, and Marcelo F. Frias. 2013. TACO: Efficient SAT-Based Bounded Verification Using Symmetry Breaking and Tight Bounds. *IEEE Trans. Software Eng.* 39, 9 (2013), 1283–1307. https://doi.org/10.1109/TSE.2013.15

[16] Juan P. Galeotti, Nicolás Rosner, Carlos López Pombo, and Marcelo F. Frias. 2010. Analysis of invariants for efficient bounded verification. In *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010*, Paolo Tonella and Alessandro Orso (Eds.). ACM, 25–36. https://doi.org/10.1145/1831708.1831712

[17] 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, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6605)*, Parosh Aziz Abdulla and K. Rustan M. Leino (Eds.). Springer, 173–188. https://doi.org/10.1007/978-3-642-19835-9_15

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

[19] Sumit Gulwani, William R. Harris, and Rishabh Singh. 2012. Spreadsheet Data Manipulation Using Examples. *Commun. ACM* 55, 8 (Aug. 2012), 97–105. https://doi.org/10.1145/2240236.2240260

[20] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence* (San Francisco, California, USA) *(AAAI'17)*. AAAI Press, 1345–1351.

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

[22] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. 2011. Automated Atomicity-Violation Fixing. *SIGPLAN Not.* 46, 6 (June 2011), 389–400. https://doi.org/10.1145/1993316.1993544

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

[24] Shadi Abdul Khalek, Guowei Yang, Lingming Zhang, Darko Marinov, and Sarfraz Khurshid. 2011. TestEra: A tool for testing Java programs using alloy specifications. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, Perry Alexander, Corina S. Pasareanu, and John G. Hosking (Eds.). IEEE Computer Society, 608–611. https://doi.org/10.1109/ASE.2011.6100137

[25] Tanvir Ahmed Khan, Allison Sullivan, and Kaiyuan Wang. 2021. AlloyFL: A Fault Localization Framework for Alloy. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1535–1539. https://doi.org/10.1145/3468264.3473116

[26] 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*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, 802–811. https://doi.org/10.1109/ICSE.2013.6606626

[27] 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.

[28] 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)*. Association for Computing Machinery, New York, NY, USA, 314–325. https://doi.org/10.1145/3338906.3338935

[29] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. JFIX: Semantics-Based Repair of Java Programs via Symbolic PathFinder *(ISSTA 2017)*. Association for Computing Machinery, New York, NY, USA, 376–379. https://doi.org/10.1145/3092703.3098225

[30] 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. https://doi.org/10.1145/3106237.3106309

[31] Xuan-Bach D. Le, Ferdian Thung, David Lo, and Claire Le Goues. 2018. Overfitting in Semantics-Based Automated Program Repair. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 163. https://doi.org/10.1145/3180155.3182536

[32] 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. https://doi.org/10.1109/TSE.2011.104

[33] 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. https://doi.org/10.1145/3377811.3380345

[34] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: Revisiting Template-Based Automated Program Repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 31–42. https://doi.org/10.1145/3293882.3330577

[35] Fan Long and Martin Rinard. 2015. Staged Program Repair with Condition Synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 166–178. https://doi.org/10.1145/2786805.2786811

[36] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 298–312. https://doi.org/10.1145/2837614.2837617

[37] 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 (Lecture Notes in Computer Science, Vol. 12071)*, Alexander Raschke, Dominique Méry, and Frank Houdek (Eds.). Springer, 61–77. https://doi.org/10.1007/978-3-030-48077-6_5

[38] 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

[39] 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, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, Laura K. Dillon, Willem Visser, and Laurie Williams (Eds.). ACM, 691–701. https://doi.org/10.1145/2884781.2884807

[40] Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. 2019. DeepDelta: Learning to Repair Compilation Errors. 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)*. Association for Computing Machinery, New York, NY, USA, 925–936. https://doi.org/10.1145/3338906.3340455

[41] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. 2016. Reducing combinatorics in GUI testing of android applications. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, Laura K. Dillon, Willem Visser, and Laurie A. Williams (Eds.). ACM, 559–570. https://doi.org/10.1145/2884781.2884853

[42] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. 2013. SemFix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*. 772–781. https://doi.org/10.1109/ICSE.2013.6606623

[43] 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.

[44] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An Analysis of Patch Plausibility and Correctness for Generate-and-Validate Patch Generation Systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 24–36. https://doi.org/10.1145/2771783.2771791

[45] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad. 2017. Elixir: Effective object-oriented program repair. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 648–659. https://doi.org/10.1109/ASE.2017.8115675

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

[47] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the Cure Worse than the Disease? Overfitting in Automated Program Repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo,

Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 532–543. https://doi.org/10.1145/2786805.2786825

[48] Armando Solar-Lezama. 2013. Program sketching. *Int. J. Softw. Tools Technol. Transf.* 15, 5-6 (2013), 475–495. https://doi.org/10.1007/s10009-012-0249-7

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

[50] 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.

[51] 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. https://doi.org/10.1109/ICST.2018.00047

[52] Allison Sullivan, Kaiyuan Wang, Razieh Nokhbeh Zaeem, and Sarfraz Khurshid. 2017. Automated Test Generation and Mutation Testing for Alloy. In *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*. IEEE Computer Society, 264–275. https://doi.org/10.1109/ICST.2017.31

[53] Shin Hwei Tan, Hiroaki Yoshida, Mukul R. Prasad, and Abhik Roychoudhury. 2016. Anti-Patterns in Search-Based Program Repair. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 727–738. https://doi.org/10.1145/2950290.2950295

[54] Emina Torlak and Daniel Jackson. 2007. Kodkod: A Relational Model Finder. In *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4424)*, Orna Grumberg and Michael Huth (Eds.). Springer, 632–647. https://doi.org/10.1007/978-3-540-71209-1_49

[55] 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*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 577–588. https://doi.org/10.1145/3238147.3238162

[56] 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*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 103–106. https://doi.org/10.1109/ICSE-Companion.2019.00049

[57] Kaiyuan Wang, Allison Sullivan, Manos Koukoutos, Darko Marinov, and Sarfraz Khurshid. 2018. Systematic Generation of Non-equivalent Expressions for Relational Algebra. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 6th International Conference, ABZ 2018, Southampton, UK, June 5-8, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10817)*, Michael J. Butler, Alexander Raschke, Thai Son Hoang, and Klaus Reichl (Eds.). Springer, 105–120. https://doi.org/10.1007/978-3-319-91271-4_8

[58] 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*, Marco Vieira, Henrique Madeira, Nuno Antunes, and Zheng Zheng (Eds.). IEEE, 391–402. https://doi.org/10.1109/ISSRE5003.2020.00044

[59] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-Aware Patch Generation for Better Automated Program Repair. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/3180155.3180233

[60] J. Xuan, M. Martinez, F. DeMarco, M. Clément, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering* 43, 1 (2017), 34–55. https://doi.org/10.1109/TSE.2016.2560811

[61] P. Zave. 2017. Reasoning About Identifier Spaces: How to Make Chord Correct. *IEEE Transactions on Software Engineering* 43, 12 (2017), 1144–1156.

[62] Guolong Zheng, Quang Loc Le, ThanhVu Nguyen, and Quoc-Sang Phan. 2019. Automatic Data Structure Repair UsingSeparation Logic. *SIGSOFT Softw. Eng. Notes* 43, 4 (Jan. 2019), 66. https://doi.org/10.1145/3282517.3282528

[63] Guolong Zheng, ThanhVu Nguyen, Simón Gutiérrez Brida, Germán Regis, Marcelo F. Frias, Nazareno Aguirre, and Hamid Bagheri. 2021. FLACK: Counterexample-Guided Fault Localization for Alloy Models. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 637–648. https://doi.org/10.1109/ICSE43902.2021.00065