BeAFix: An Automated Repair Tool for Faulty Alloy Models

Simón Gutiérrez Brida*†, Germán Regis*, Guolong Zheng‡,
Hamid Bagheri‡, ThanhVu Nguyen‡, Nazareno Aguirre*†, Marcelo Frias†§
*Department of Computer Science, FCEFQyN, University of Río Cuarto, Argentina
†National Council for Scientific and Technical Research (CONICET), Argentina
‡Department of Computer Science & Engineering, University of Nebraska-Lincoln, USA
§Department of Software Engineering, Buenos Aires Institute of Technology, Argentina

***Abstract*—This paper describes BeAFix, a tool for automated repair of faulty Alloy models. The tool builds upon the Alloy Analyzer, the analysis tool for Alloy. It generates repair candidates by mutating a faulty Alloy model, and employs a bounded-exhaustive approach to traverse the space of repair candidates. Since BeAFix's mutation operators make the space of repair candidates to quickly grow, the tool supports some sound pruning techniques, that allow it to fix Alloy models with more than one faulty line or expression. Additionally, BeAFix does not require tests as a patch acceptance criterion. Although BeAFix supports tests as oracles, our tool is also able to leverage property-based oracles, which are more commonly found in Alloy models in the form of predicate satisfiability and assertion validity checks. A video demonstration of BeAFix can be found at https://youtu.be/5RG40SmlFXQ. The tool's binaries and further details about its usage, can all be found at http://sites.google.com/view/beafixevaluation.**

***Index Terms*—Alloy, Automated Repair, Bounded-Exhaustive Analysis**

I. Introduction

The problem of guaranteeing that a software system behaves correctly according to the user expectations is still one of the most challenging problems in software engineering, despite the significant advances in techniques for software validation and verification. Ambiguities and incompleteness issues in software requirements, among other issues introduced as part of early software design ideas, are common causes of unexpected behaviors being observed in delivered software. Formal notations for capturing software requirements and stating abstract characterizations of the software-to-be, can help in reducing these issues. More precisely, by analyzing abstract formal models of software, one can discover missing problem domain constraints or flaws in abstract software designs, anticipating problems long before the software is developed.

Alloy [2] is a formal specification language, well suited to aid in the above described situations, because it features a fully automated SAT-based specification analysis mechanism. Alloy's language is expressive, based on a first-order relational logic that includes transitive closure. Its specifications can be automatically analyzed using Alloy Analyzer, the language's main tool. The analyses that this tool supports are predicate satisfiability checks (checking whether a given property is consistent with the specification constraints) and assertion validity checks (checking whether a given property is a consequence of the specification constraints). Both analyses are *bounded*, or more precisely, bounded-exhaustive: given a user provided bound (maximum number of elements in the specification's domains), the tool either finds a property to be satisfiable (resp., valid) within the bound, or guarantees that no model (resp., counterexample) of the property, within the bound, exists. These checks are implemented through a reduction to SAT solving.

As with any formal notation, correctly modeling a given software design and corresponding problem domain may not be straightforward. Developers can make mistakes, formalization mistakes in particular, i.e., mistakes associated with wrongly expressing, in the formal notation, the developers' intention. These mistakes lead to faulty Alloy specifications, which many times can be identified via Alloy's analyses, when analysis results differ from what the developers expected. For instance, the developer may have expected a predicate to be consistent with the specification and is not, or an assertion to follow from the specification, and it does not.

In this paper, we present BeAFix, a tool that aids in debugging faulty Alloy specifications by performing automated repair. The tool is similar in spirit to automated program repair tools, but for the context of formal specifications instead of programs. As explained in [1], where the tool's technique is introduced, this is not just a simple change of context: certain characteristics make programs and specifications different, and techniques for program repair difficult to adapt to the specification context. For instance, in programming contexts one often finds test cases, which can be used as patch acceptance criterion as well as to drive the repair process to the most suspicious parts of a program. But tests are not naturally found accompanying formal specifications. Also, large software repositories can be the source of patch patterns, that techniques can identify and use to repair other programs. But no such repositories are available for formal software specifications.

BeAFix takes advantage of common Alloy specification elements. Rather than using test cases as patch acceptance criteria, BeAFix attempts to exploit the property-based oracles that are typically found in Alloy specifications, such as predicates and assertions associated with analyses (satisfiability/validity checks). These properties are usually more general than

specific test cases, and therefore naturally constitute stronger patch acceptance criteria, diminishing the impact of overfitting. Moreover, BeAFix follows a *bounded-exhaustive* approach to specification repair: given a faulty Alloy model, a set of suspicious locations, a set of expression mutation operators and a bound k, the tool either finds a fix, i.e., a modification of the suspicious locations that applies the mutation operators at most k times per location, or it guarantees that no such fix is possible. Thus, the technique fits well with the kind of analysis that the Alloy user is accustomed to.

BeAFix supports Alloy specifications with multiple buggy expressions. It also supports different repair "oracles", in particular failing predicates and assertions, as well as the recently proposed Alloy unit tests [5]. However, the tool does not perform fault localization: it is a task that is performed only once, before the repair process starts, and can be delegated to Alloy fault localization tools [7], [8]. Our presentation concentrates on the tool itself. Details on the repair technique, and in particular on its sound pruning approaches (necessary to help with the tool's efficiency, as the space of repair candidates is bounded exhaustively explored) are only briefly described, as these are part of the technical paper that introduced the technique [1].

II. Using BeAFix

BeAFix is implemented as an extension of the Alloy Analyzer. The Alloy user will therefore feel comfortable with the tool. All standard Alloy Analyzer functionalities for editing and analyzing specifications are exactly as in the original tool. Features specific to specification repair appear as additional options, that do not affect the standard options.

In order to describe how the tool is used, let us consider an abstract formal specification of a file system. The specification is originally introduced in [4]. The objects of a file system can be *files* or *directories*, which are specified as *signatures* (the mechanism to define data domains in Alloy). These signatures are combined in an additional one, `FSObject`, which the former extend (Alloy allows one to define data domain inclusion via signature extension, and signatures which are solely composed of their extensions via the "abstract" keyword). A file system is defined using a structured signature, with fields that capture its set of `live` objects, a distinguished `root` directory, a `parent` relation stating that every element of the system, except the root, belongs to exactly one directory, and the relation that represents the directories' `contents`. Specification assumed constraints are specified in Alloy via *facts*, expressed in Alloy's relational logic. This specification's assumptions state that all `live` system objects can be reached from the `root`, recursively via the `contents` relation; also, the `parent` relation is the transpose of `contents`. This part of the specification is then as follows:

```
1  abstract sig FSObject { }
2  sig File, Dir extends FSObject { }
3  sig FileSystem {
4    live: set FSObject,
5    root: Dir & live,
6    parent: (live - root) -> one (Dir & live),
7    contents: Dir -> FSObject
8  }
9  {
10   live in root.*contents
11   parent = ~contents
12 }
```

Now that we have the structural specification of the file systems, we would like to capture some operations on file systems, such as moving an element from a directory to another, or recursively removing all elements contained in a given one. Model operations are specified in Alloy via predicates. A `move` predicate will capture the move operation, and a `removeAll` predicate the recursive removal. Predicate parameters represent the file system an operation is applied on, the resulting system (if the operation may modify it), as well as other operation arguments and return values. As a convention, primed variables represent resulting values. For instance the `move` operation will apply to a `fs` file system, it will "return" a `fs'` file system, and needs to receive both the object to be moved, and the target directory. The body of the predicate captures the behavior of the operation as a relationship between the corresponding parameter variables. The operations are then specified as follows:

```
12 pred move [fs, fs':FileSystem, x:FSObject, d:Dir]{
13   (x + d) in fs.live
14   fs'.parent = fs.parent- x->(x.(fs.parent))- x->d
15 }
16
17 pred removeAll [fs, fs': FileSystem, x: FSObject]{
18   x in (fs.live - fs.root) and fs'.root = fs.root
19   let subtree = x.*(fs.contents) |
20    fs'.parent =
21     fs.parent - subtree->(subtree.(fs.contents))
22 }
```

While it is not obvious, both operations are wrongly specified. The `move` operation should state that `d` is the new parent of `x`, and it does not (it actually "removes" this map). The `removeAll` operation, on the other hand, wrongly specified how the `parent` relation is updated (removing the containment in descendants of `subtree`, rather than ancestors of it).

As proposed in [4], one can check the specification by means satisfiability checks, and by defining assertions that capture intended properties of the specification, in particular of the two operations. For the `move` operation, the assertion `moveOkay` states that the operation should not alter the objects of the file system. For the `removeAll` operation, the `removeAllOkay` assertion states that the operation removes the direct and indirect contents of the removed element. These assertions look as follows:

```
21 assert moveOkay {
22   all fs, fs': FileSystem, x: FSObject, d:Dir |
23     move[fs, fs', x, d] => fs'.live = fs.live
24 }
25
26 assert removeAllOkay {
27   all fs, fs': FileSystem, d: Dir |
28     removeAll[fs, fs', d] =>
29       fs'.live = fs.live - d.*(fs.contents)
30 }
31
```

```
run move for 5 expect 1
run removeAll for 5 expect 1
check moveOkay for 5
check removeAllOkay for 5
```

Four analysis commands define the expectations on the specification: predicates `move` and `removeAll` are expected to be consistent with the specification, and assertions `moveOkay` and `removeAllOkay` are expected to hold. When these are checked, Alloy Analyzer determines that the assertions are invalid, and produces witnessing counterexamples. At this point, we can use BeAFix to attempt to repair the specification, i.e., to modify it so that all analysis commands succeed. We refer to these expectations as the repair *oracle*.

To perform the repair process, BeAFix requires the provision of one or more suspicious locations, i.e., expressions in the specification that are assumed to be the cause of the defect in the specification. These suspicious locations can be whole formulas, lines, or other subexpressions within the specification. However, signature definitions, that define the domains of the specification and its structure, cannot be marked as suspicious. The suspicious locations may be manually indicated, or automatically generated, by resorting to some fault localization technique for Alloy, such as FLACK [8] or AlloyFL [7]. As we illustrate in Figure 1, these locations are indicated in the specification text by enclosing the corresponding expression together with the symbol **#m#** ❶. One can then execute the *Repair by mutating marked exprs #m#* from the *Execute* Alloy Analyzer menu ❷.

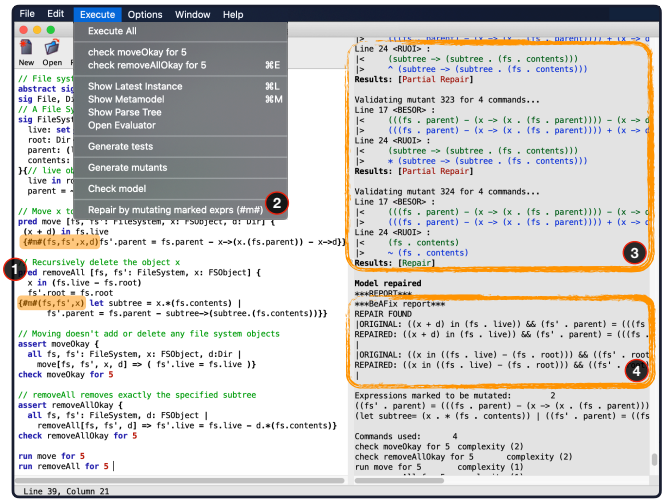

Fig. 1. BeAFix Graphical User Interface.

During repair, the GUI version of the tool displays the fix candidates being considered (both the original expression and corresponding mutated expression, for reference), and the analysis verdicts (which parts of the oracles passed and failed, etc.) ❸. A detailed report is shown when the repair process finishes, and if a fix is found, the proposed changes are reported ❹, and the corresponding fixed version of the specification is written to a file. For our running example, BeAFix found a fix with the following modifications:

```
Line 14:
ORIGINAL: ...(x -> (x.(fs.parent)))) - (x -> d))
REPAIRED: ...(x -> (x.(fs.parent)))) + (x -> d))

Line 21:
ORIGINAL: ...(subtree -> (subtree.(fs.contents))))
REPAIRED: ...(subtree -> (subtree.~(fs.contents))))
```

BeAFix allows the user to set various parameters, including a *maxdepth* (maximum number of mutations allowed per marked expression) and a *timeout* (time budget, whose default is "unlimited"). The two pruning techniques, *variabilization* and *partial repair*, can be enabled or disabled for the repair process. Pruning techniques only work when the faulty specification has more than one suspicious location. Further details on how the pruning techniques operate, and precisely when these apply, can be found in [1]. The effect of pruning can be assessed by running the repair process with it enabled and disabled. For instance, in our running example, repairing without pruning explores more than 4600 repair candidates before finding a fix, while the tool only explores 324 candidates when pruning is enabled.

BeAFix can be run from the command line too; we provide a specific version of the tool for this usage.

III. How BeAFix Works

BeAFix does not perform fault localization. This task is delegated either to the user, or to an external tool. BeAFix then starts with an Alloy specification with some marked suspicious expressions. Assuming that the specification is faulty, i.e., at least one of the analysis commands fails, BeAFix will attempt to fix the specification by mutating the marked expressions using a set of expression mutation operators, with the aim of finding a modification of the specification where all commands succeed. BeAFix's repair algorithm is graphically depicted in Figure 2. The algorithm maintains a queue of candidate fixes, initialy populated with only the original faulty specification. In each iteration, the first candidate is dequeued and the analysis commands are checked on it to validate it; if at least one command fails, then all applicable mutations on this candidate are applied (as long as the max. mutation depth is not reached) and the results enqueued, before discarding the failing candidate. The candidate offspring generation involves the sound pruning strategies, that may lead to discarding some of the candidates (when pruning applies, these candidates are not even generated, of course). The process is iterated until a fix is found (all analysis commands succeed on the current fix candidate), the queue becomes empty, or the time budget for analysis is reached.

The pruning techniques that BeAFix supports are *variabilization* and *partial repair*. Both are concerned with specifications that have more than one suspicious location. Variabilization performs a check of a candidate expression e', replacing a suspicious expression e, in combination with the remaining faulty locations in the specification. Intuitively, it performs a check to analyze whether leaving e' fixed (i.e.,

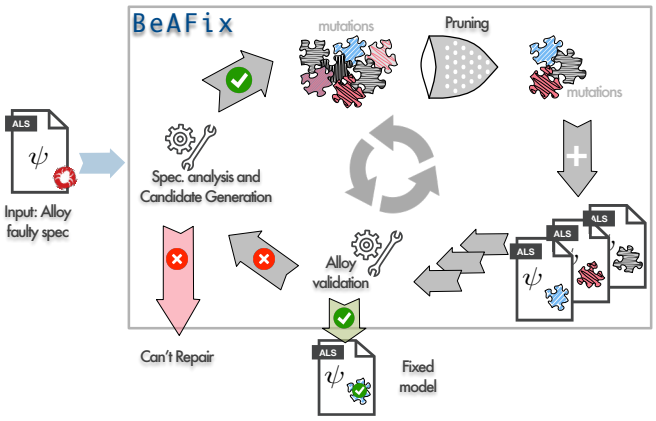

Fig. 2. BeAFix repair process.

removing it from the suspicious locations) may lead to a repair or not; if fixing e' necessarily leads to violating a command, then e', and its combination with mutations for the remaining locations, are disregarded. Partial repair, on the other hand, performs a syntactic analysis between the dependencies of the commands in a given candidate and the suspicious locations, to decide whether a particular fix candidate can be considered or disregarded, without necessarily checking all commands. Both pruning techniques are sound, in the sense that they can only disregard fix candidates that would not pass the analysis checks, and complement each other.

IV. Final Remarks

Formal specification practitioners are aware of the subtleties of correctly capturing, in a formal notation, a given abstract design and the relevant characteristics of the corresponding problem domain. Tools like Alloy have had great success in formal specification contexts, among other reasons, because of its outstanding automated analysis support, that can help in incrementally developing formal models, fixing issues, adding missing assumptions, and adapting design decisions. This process has been largely manual, and it has recently received attention with repair tools such as ARepair [6], and tools for automating auxiliary tasks such as fault localization.

Our tool BeAFix arrives to this context, and attempts to profit from standard Alloy "oracles", which are naturally stronger than test cases, to effectively suggest fixes to faulty Alloy specifications. As a consequence, the tool is less prone to overfitting, compared to test-based techniques. As the evaluation in [1] shows, BeAFix was able to effectively repair an important part of a benchmark of over two thousand faulty models, with significantly less overfitting than alternative techniques. It can also correctly repair an important number of cases in the ARepair benchmark (the benchmark introduced in [6]), without the need to manually craft test cases. Figure 3 shows an overview of these results.

The experimental dataset that was used to assess BeAFix was built by complementing the ARepair benchmark, with a

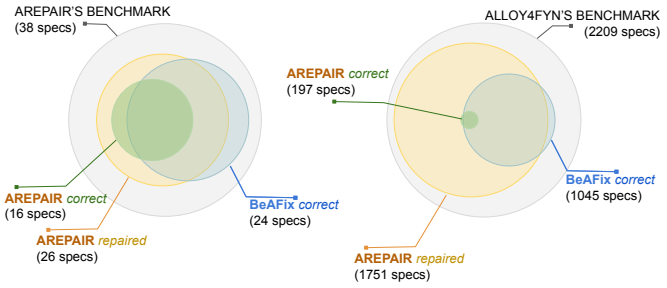

Fig. 3. BeAFix evaluation

set of real faulty specifications originated in the Alloy4Fun project [3]. This project gathers a large collection of information of student sessions solving specification assignments. We inspected and combined these sessions to construct specifications performed by a same student, accumulating different specification defects. The resulting faulty specification dataset has, as we mentioned, more than two thousand faulty Alloy models. The tool, the experimental dataset, and the scripts to reproduce the experiments, are all publicly available in:

https://sites.google.com/view/beafixevaluation/

References

[1] Simón Gutiérrez Brida, Germán Regis, Guolong Zheng, Hamid Bagheri, ThanhVu Nguyen, Nazareno Aguirre, and Marcelo F. Frias. Bounded exhaustive search of alloy specification repairs. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pages 1135–1147. IEEE, 2021.

[2] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[3] Nuno Macedo, Alcino Cunha, José Pereira, Renato Carvalho, Ricardo Silva, Ana C. R. Paiva, Miguel Sozinho Ramalho, and Daniel Castro Silva. Experiences on teaching alloy with an automated assessment platform. In Alexander Raschke, Dominique Méry, and Frank Houdek, editors, *Rigorous State-Based Methods - 7th International Conference, ABZ 2020, Ulm, Germany, May 27-29, 2020, Proceedings*, volume 12071 of *Lecture Notes in Computer Science*, pages 61–77. Springer, 2020.

[4] Rob Seater and Greg Dennis. http://alloytools.org/tutorials/online/index.html.

[5] Allison Sullivan, Kaiyuan Wang, and Sarfraz Khurshid. 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*, pages 398–403. IEEE Computer Society, 2018.

[6] Kaiyuan Wang, Allison Sullivan, and Sarfraz Khurshid. Automated model repair for alloy. In Marianne Huchard, Christian Kästner, and Gordon Fraser, editors, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 577–588. ACM, 2018.

[7] Kaiyuan Wang, Allison Sullivan, Darko Marinov, and Sarfraz Khurshid. Fault localization for declarative models in alloy. In Marco Vieira, Henrique Madeira, Nuno Antunes, and Zheng Zheng, editors, *31st IEEE International Symposium on Software Reliability Engineering, ISSRE 2020, Coimbra, Portugal, October 12-15, 2020*, pages 391–402. IEEE, 2020.

[8] Guolong Zheng, ThanhVu Nguyen, Simón Gutiérrez Brida, Germán Regis, Marcelo F. Frias, Nazareno Aguirre, and Hamid Bagheri. Flack: Counterexample-guided fault localization for alloy models. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 637–648, 2021.