Artifact of Bounded Exhaustive Search of Alloy Specification Repairs

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*—BeAFix is a tool and technique for automated repair of faulty models written in Alloy, a declarative formal specification language based on first-order relational logic. BeAFix takes a faulty Alloy model, i.e., an Alloy model with at least one analysis command whose result is contrary to the developer's expectation, and a set of suspicious specification locations, and explores the space of fix candidates consisting of all alternative expressions for the indicated locations, that can be constructed by bounded application of a family of mutation operations. BeAFix can work with any kind of specification oracle, from Alloy test cases to standard predicates and assertions typically found in Alloy specifications, and is backed with a number of sound pruning strategies, for efficient exploration of fix candidate search spaces.**

I. Introduction

Formal specifications have many applications in software development. They can be used to precisely capture software requirements as well as constraints and assumptions of the problem domain, to express abstract formulations of software designs, and to formally specify program properties such as postconditions and invariants, among other uses. The precise semantics associated with a formal notation makes it suitable for automated analysis, that can help discover flaws in software artifacts specified in the notation. The Alloy specification language [1] is a formal specification language that exploits this observation, and supports the automated analysis of specifications via bounded validity and satisfiability checking, based on SAT solving.

Correctly modeling a software artifact using a formal notation such as Alloy can be challenging, and subtle errors due to the incorrect use of the language may arise, even from experienced users. Thus, techniques for automated repair, such as those commonly found in the context of programming languages, are also relevant for formal specification languages. We present BeAFix, a tool (and technique) for automated repair of Alloy specifications, with some distinguishing features. Firstly, the tool supports any kind of specification oracle, from Alloy test cases, to standard predicates or assertions, typically found as part of Alloy specifications. Secondly, BeAFix's repair approach is *bounded exhaustive*, in the sense that, given a number of suspicious locations, it considers all possible fix candidate expressions that can be obtained by applying mutations to the suspicious expressions, up to a given bound. Thus, when the tool terminates, it either finds a fix, or guarantees that no fix is possible by mutating the suspicious locations up to the given bound, and with the provided mutation operations. Alloy users are accustomed to bounded exhaustive analyses, as this is the kind of analysis associated with Alloy Analyzer, the tool for automatically checking for bounded satisfiability or bounded validity of properties. Since the space of fix candidates can be very large for specifications with multiple suspicious locations, our tool comes equipped with a number of sound pruning techniques, that improve its efficiency in the automated repair process.

II. Using BeAFix

BeAFix provides two different interfaces. Its *command line interface* is better suited for batch specification repair, for pipelining the tool with other techniques (e.g., for fault localization), and for writing scripts to reproduce experiments. The *graphical user interface*, on the other hand, offers a more natural front-end for Alloy users, as it extends the Alloy Analyzer, preserving its look and feel. The graphical user interface also provides some further runtime information during the repair process, such as the current fix candidate being considered, etc.

Our tool receives a faulty Alloy specification, i.e., an Alloy specification with at least one analysis command whose outcome is contrary to the corresponding expectation (e.g., a predicate expected to be satisfiable, which Alloy Analyzer deemed unsatisfiable). BeAFix also requires the provision of one or more suspicious locations, i.e., parts of the specification that are assumed to be the cause of the defect in the specification. These suspicious locations can mark formulas, lines, or other subexpressions within the specification. However, signature definitions, that define the domains of the specifications and their structure, cannot be marked as suspicious. The suspicious locations may be manually indicated by the developer, or automatically generated, by resorting to some fault localization technique for Alloy, such as FLACK or AlloyFL. Finally, the tool also receives a *bound*, the maximum number of mutations that can be applied to each suspicious location.

Suspicious locations are indicated in the Alloy specification via a special *mark*, `#m#`. The marked expression can declare context variables, which, intuitively, indicate certain bound

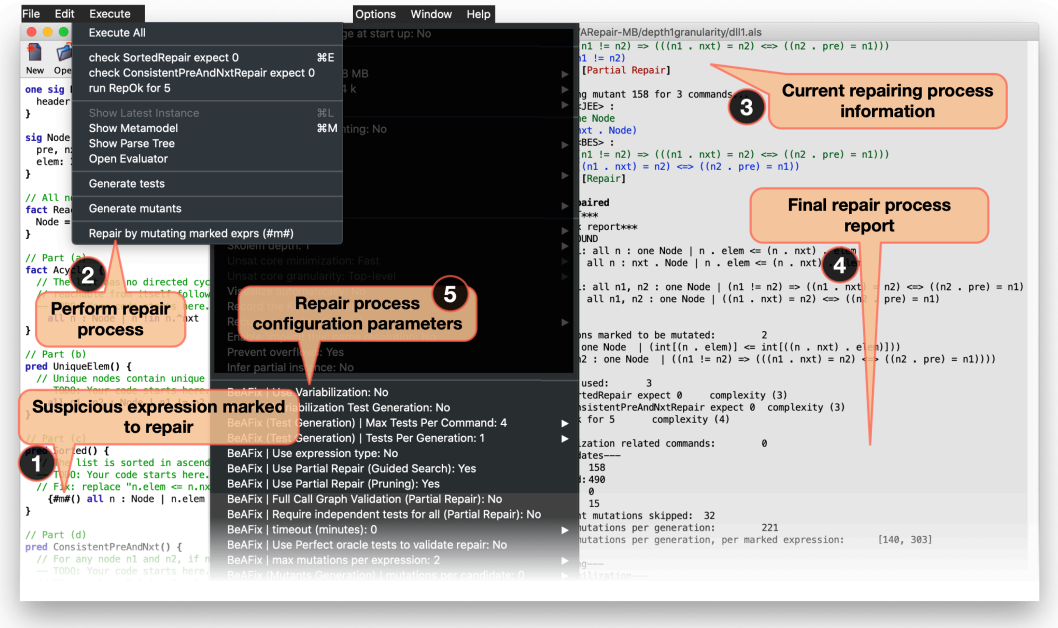

Fig. 1. BeAFix Graphical User Interface

variables that the suspicious expression may depend on. This is in fact related to one of our pruning strategies, *variabilization*, and we refer the reader to our technical paper [2] for further details. For example, the marked expression `{#m#() some x : T | P[x]}` considers the whole quantified formula as a suspicious location, that BeAFix will mutate to produce fix candidates (the quantifier, the type of the quantified variable and the body of the formula, can all be mutated). Alternatively, the expression `some x : T | {#m#(x) P[x]}` marks the body of the quantified formula as suspicious, with `x` being a context variable for the suspicious location; in this latter case, BeAFix will only mutate the body of the quantification, preserving as is both the quantifier and the type of variable `x`.

Our tool 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 and disabled as well (disabled by default). 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 [2].

BeAFix is a Java application. Both the command line and the GUI versions of BeAFix are distributed as jar bundles, which can be found in the tool's repository[1], together with detailed instructions on how to install and run the tool. The tool and replication package are also available at a public archive[2]. The command line version receives all the parameters directly from the command line. The GUI version of BeAFix is an extension of the Alloy Analyzer. Figure II summarizes the additional features that BeAFix provides, with respect to Alloy Analyzer. During repair, the GUI version 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.). A detailed report is shown when the repair process finishes, and if a fix is found, it is written to a file. The repaired specification may contain redundant parentheses, due to limitations in our current implementation of the specification writer.

[1]https://github.com/gregistecco/BeAFixICSE2021ArtifactEval

[2]https://doi.org/10.6084/m9.figshare.13626776.v2

III. Replicating Experiments

The experimental evaluation in [2] can be replicated by installing BeAFix natively and downloading the case studies, or using a Docker container, that we have prepared for the convenience of the user. The Docker container includes the tool already configured, the case studies, and scripts for running the experiments. Instructions on how to replicate the experiments, either natively or using the Docker container, can be found in our tool's repository. Notice that the number of repair subjects is rather large, and thus reproducing all the experiments in batch mode can take many hours. We provide scripts to run experiments separately, or grouped by case study, for convenience.

References

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

[2] S. Gutiérrez-Brida, G. Regis, G. Zheng, H. Bagheri, T. Nguyen, N. Aguirre, and M. Frias, "Bounded exhaustive search of alloy specification repairs," in *Proceedings of the 43rd ACM/IEEE International Conference on Software Engineering ICSE 2021, Virtual (originally Madrid, Spain), 23-29 May 2021*, 2021.