Poster: Program Repair That Learns From Mistakes

Bat-Chen Rothenberg*
Technion - Israel Institute of Technology
batg@cs.technion.ac.il

Orna Grumberg
Technion - Israel Institute of Technology
orna@cs.technion.ac.il

ABSTRACT

Automated program repair is a very active research field, with promising results so far. Several program repair techniques follow a Generate-and-Validate work-scheme: programs are iteratively sampled from within a predefined repair search space, and then checked for correctness to see if they constitute a repair.

In this poster, we propose an enhanced work-scheme, called *Generate-Validate-AnalyzeErr*, in which whenever a program is found to be incorrect, the error trace that is the evidence of the bug is further analyzed to obtain a *search hint*. This hint improves the sampling process of programs in the future. The effectiveness of this work-scheme is illustrated in a novel technique for program repair, where search hints are generated in a process we call *error generalization*. The goal of error generalization is to remove from the search space all programs that exhibit the same erroneous behavior.

The aim of this poster is to present our vision of the future of program repair, and trigger research in directions that have not been explored so far. We believe that many existing techniques can benefit from our new work-scheme, by focusing attention on what can be learned from failed repair attempts. We hope this poster inspires others and gives rise to further work on this subject.

ACM Reference Format:
Bat-Chen Rothenberg and Orna Grumberg. 2018. Poster: Program Repair That Learns From Mistakes. In *ICSE '18 Companion: 40th International Conference on Software Engineering Companion, May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3183440.3195104

1 INTRODUCTION

In the process of software production and maintenance, much effort is invested in order to ensure that the product is as bug free as possible. Manual bug repair is time-consuming and requires close acquaintance with the checked program. Therefore, in recent years there has been much progress in the development of tools for automated program repair.

Existing program repair techniques look for a repair to a program P by making changes to P according to a predefined set of rules. We refer to the set of programs created in this way as the *search space* of the technique, and to any program in it as a *candidate program*. Many techniques ([2–5]) use a *Generate-and-Validate* working scheme. This means that each time a candidate program is sampled according to a certain policy, and the correctness of this program is examined. If it is correct - it is returned as a repair, otherwise - another candidate program is chosen, and so on.

In this work, we propose an enhanced working scheme, called *Generate-Validate-AnalyzeErr*, in which whenever a program is found to be incorrect, a witness of the bug is further analyzed to obtain a *search hint*, which improves the sampling of candidate programs in the future. We call it "repair that learns from mistakes".

2 GENERATE-VALIDATE-ANALYZE ERROR

An illustration of the Generate-and-Validate work-scheme is presented in Figure 1a. This scheme begins with the Generate stage, where a candidate program P' is sampled according to a certain policy. Then, the Validate stage checks if that program is correct or not. If it is correct, then it is returned as a repair. Otherwise, the Generate stage samples a different program, and the interplay continues.

In the simple Generate-and-Validate scheme, the output of the Validate stage is a plain yes-or-no answer. However, in cases where the Validate stage is realized by running all tests in a test suite, a "no" answer is coupled with a witness in the form of an *error trace*. An error trace is a triple (I, π, Σ), where I is an input to P' leading to an output different than expected in the test suite, π is the path (i.e., sequence of statements) this input follows during execution, and Σ is the sequence of program states (i.e., variable valuations) at every point along π. [1] In fact, even if the Validate stage is realized by a formal verification tool checking the validity of some safety property, most such tools also return an error trace in case the property is violated (in which case the input causes the violation of the property).

Our main insight is that the knowledge embedded in error traces is not exploited in the traditional Generate-and-Validate work-scheme. Therefore, we propose a novel work-scheme, which we call Generate-Validate-AnalyzeErr (cf. Figure 1b). In our new scheme, an additional stage named *AnalyzeErr* is inserted after the Validate stage, before returning to the Generate stage. Whenever a candidate program P' is found incorrect, the witness error trace is passed on to the AnalyzeErr stage. There, further analysis is carried out, and the result is a *search hint* that is passed on to the Generate stage.

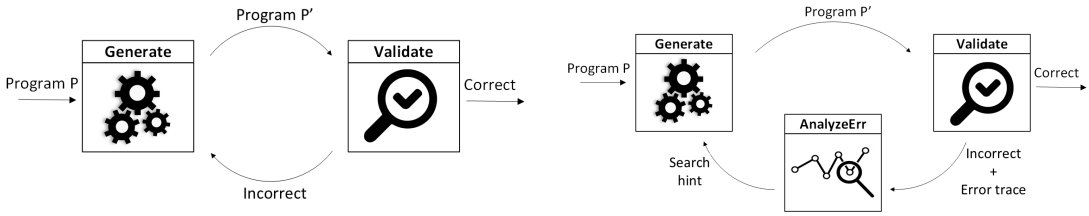
We consider a search hint to be any information that can be extracted from an error trace and used to facilitate or improve the Generate stage. For example, for an error trace (I, π, Σ), a useful search hint can be a set of programs that contain a bug on the input I. Such a set is useful, as it can be pruned from the search space

*Work of this author was partially supported by the Technion Hiroshi Fujiwara cyber security research center and the Israel cyber bureau.

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5663-3/18/05.
https://doi.org/10.1145/3183440.3195104

[1] For some applications of our work-scheme, π and Σ might not be necessary and may be omitted from the definition.



(a) Generate-and-Validate **(b) Generate-Validate-AnalyzeErr**

Figure 1: work-schemes for program repair

without further inspection. One might be interested in a more light-weight AnalyzeErr stage, at the expense of possibly losing valid repairs. In this case, a search hint can eliminate from the search space a set of programs that are only likely to contain a bug, but are not guaranteed to do so. The option to fine tune the level of effort required during AnalyzeErr is one of the great advantages of our work-scheme.

In methods where the Generate stage is based on ranking all candidate programs and iteratively choosing the best one, a search hint can also be used to improve ranking on-the-fly. For example, a search hint can be a set of program locations that are more suspicious than others. Fine tunning of the analysis level is possible also in this example; one may simply return the set of statements in π, or apply different methods for fault localization, e.g., [1], to obtain a smaller subset of it.

The above examples are just the tip of the iceberg of exploiting search hints for program repair. We believe that the development of useful, sophisticated search hints can be a fruitful research topic. We hope this poster inspires others and gives rise to further work on this subject.

3 ERROR GENERALIZATION FOR MUTATION-BASED PROGRAM REPAIR

To demonstrate how beneficial Generate-Validate-AnalyzeErr can be, we have extended our program repair method [5], which follows Generate-and-Validate, to Generate-Validate-AnalyzeErr. The additional AnalyzeErr stage is based on a novel error explanation method we have developed, called *error generalization*. The implementation of the improved algorithm is currently work-in-progress, but we expect to see a significant boost in performance.

Mutation-based program repair. In [5], we considered repair of C programs with respect to assertions in the code. Repair was done using a predefined set *Mut* of *mutations*, which are syntactic rules for replacement of expressions in the program (e.g., replace + with -). The search space was created by applying mutations from this set, anywhere in the program (including the possibility to apply multiple mutations to different locations at once).

Our approach followed a simple Generate-and-Validate scheme, using an interplay between a SAT solver and an SMT solver. The former was used for the generate stage; we have constructed a boolean formula where each satisfying assignment corresponds to a program in the search space. The later was used for the validate stage; a program was represented using an SMT formula which is satisfiable iff the program has a bug.

Error generalization. Next, we provide a brief overview of our error generalization algorithm. Given an error trace (I, π, Σ) and the set of mutations *Mut*, we denote $\pi = s_0, .., s_{n-1}$, $\Sigma = \sigma_0, .., \sigma_n$ and $Mut(s)$ is the set of statements obtained by applying mutations from *Mut* to s . The search hint provided by error generalization is a set of paths, Π, all ending in an assertion violation on input I (in particular, $\pi \in \Pi$). This way, even though the bug was found in a specific program, several candidate programs can be removed from the search space (all those programs that contain a path in Π).

For the generalization process, we use *state formulas*. A state formula is a first-order formula over the variables of the program; a satisfying assignment of it corresponds to a valuation of program variables, i.e., a state of the program. We generalize π to Π by generalizing each state σ_i to a state formula, φ_i. In particular, the generalization φ_n of σ_n represents only states violating the assertion.

Given the sequence of state formulas $\varphi_0 \dots \varphi_n$, we say that a program statement s is *prohibited* in location i, if whenever the program is in a state satisfying φ_i, the execution of s will lead to a state satisfying φ_{i+1}. The search hint Π is the set of all paths $s'_0, .., s'_{n-1}$, where for all i, $s'_i \in Mut(s_i)$ and s'_i is prohibited in i.

The wisdom is in computing the sequence $\varphi_0 \dots \varphi_n$ so that as many statements are prohibited in as many locations as possible. Our algorithm computes this sequence in a backwards fashion; first φ_n is set to the negation of the assertion, and then φ_i is iteratively computed from φ_{i+1}, while taking into consideration $Mut(s_i)$. Backwards computation of formulas is done using a weakest-precondition computation. Abstraction techniques can be used to increase efficiency at the expense of making Π smaller.

REFERENCES

[1] Evren Ermis, Martin Schaf, and Thomas Wies. 2012. Error invariants. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 7436 LNCS (2012), 187–201. https://doi.org/10.1007/978-3-642-32759-9_17

[2] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A generic method for automatic software repair. *Software Engineering, IEEE Transactions on* 38, 1 (2012), 54–72.

[3] Fan Long and Martin Rinard. 2015. Prophet: Automatic patch generation via learning from successful patches. (2015).

[4] Urmas Repinski, Hanno Hantson, Maksim Jenihhin, Jaan Raik, Raimund Ubar, Giuseppe Di Guglielmo, Graziano Pravadelli, and Franco Fummi. 2012. Combining dynamic slicing and mutation operators for ESL correction. In *Test Symposium (ETS), 2012 17th IEEE European*. IEEE, 1–6.

[5] Bat-Chen Rothenberg and Orna Grumberg. 2016. Sound and Complete Mutation-Based Program Repair. In *FM 2016: Formal Methods: 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings 21*, Vol. 9995. https://doi.org/10.1007/978-3-319-48989-6