Failure-Driven Program Repair

Davide Ginelli
University of Milano - Bicocca, DISCo
Milan, Italy
davide.ginelli@unimib.it

ABSTRACT

Program repair techniques can dramatically reduce the cost of program debugging by automatically generating program fixes. Although program repair has been already successful with several classes of faults, it also turned out to be quite limited in the complexity of the fixes that can be generated.

This Ph.D. thesis addresses the problem of cost-effectively generating fixes of higher complexity by investigating how to exploit failure information to directly shape the repair process. In particular, this thesis proposes Failure-Driven Program Repair, which is a novel approach to program repair that exploits its knowledge about both the possible failures and the corresponding repair strategies, to produce highly specialized repair tasks that can effectively generate non-trivial fixes.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Automatic program repair, software defects, automatic debugging

ACM Reference Format:
Davide Ginelli. 2019. Failure-Driven Program Repair. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3338906.3341464

1 RESEARCH PROBLEM

Debugging and fixing faults are time-consuming tasks that can account for a relevant portion of the development process, as reported in multiple studies [3, 20]. Indeed, debugging and fixing software faults require analyzing the failures, understanding their causes, locating faults, designing appropriate fixes, and validating the patched code. These activities must be repeated every time a new problem is investigated, regardless of the nature of the fault.

While some techniques can be exploited to partially automate some steps of the debugging process [18, 24], a promising direction is given by program repair techniques [5, 16], which can potentially deliver a fully automated program repair process, drastically reducing the effort needed to debug and repair programs. For instance, program repair techniques might be used to automatically repair or suggest repairs for a subset of the faults, while letting the developers focus their effort on the faults that cannot be addressed automatically.

Although program repair techniques have been already successful with several classes of faults, they also turned out to be quite limited in the complexity of the faults that can be repaired. In many cases the fixes consist of single or few statements that are modified relying on the plastic surgery hypothesis [2], which assumes that the statements necessary to fix a fault can be found in the faulty program itself, or using program synthesis techniques [17, 23], which are often limited to the generation of simple expressions. Techniques working with fix patterns learned from version histories can sometimes produce larger fixes [8–10], but their effectiveness is limited by the set of samples used to infer the templates.

In this context, the *research challenge* addressed by this Ph.D. thesis is to increase the complexity and size of the fixes that can be generated automatically, dramatically improving the effectiveness of automatic program repair techniques. The *research hypothesis* investigated in this work is that failures usually disclose semantic information that has been partially exploited so far and that can be extremely useful to organize effective ad-hoc program repair processes. For instance, a program that fails with an `ArrayIndexOutOfBoundsException` or with an out of range value checked by an assertion would be investigated differently by a developer.

Indeed, the `ArrayIndexOutOfBoundsException` suggests that an index used in a statement that accesses an array location may have a wrong value, that the wrong variable may have been used as index of the array, or that the wrong array may have been used. The value of a variable outside the expected range suggests that an erroneous value might have been computed for that variable or the correct computation might have been performed an erroneous number of times. The intuition is that instead of having a generic repair process that is supposed to fit all the faults, the *repair process can be defined ad-hoc* based on the observed failure and the guesses about its likely causes.

In order to achieve the ability to formulate a repair process specific to an observed failure, the repair technique must be aware of the different failure types that can be observed and known strategies to address them. Because of this knowledge embedded in the technique, I called this new approach to the automatic program repair problem *Failure-Driven Program Repair*. The intuition is that a repair strategy optimized on certain classes of failures can produce more sophisticated fixes than state of the art approaches.

The key contributions that will be delivered by this thesis are: (i) the definition of Failure-Driven Program Repair, that is, a new framework to elaborate program repair tasks dynamically based on the observed failures, (ii) the demonstration of the broad applicability of the framework, (iii) a study of the impact of the deployment architecture (e.g., sequential vs parallel) on the effectiveness of the framework, (iv) empirical evidence demonstrating effectiveness, efficiency and flexibility of Failure-Driven Program Repair.

In the rest of this paper, I discuss related work in automatic program repair (Section 2), illustrate the concept of Failure-Driven Program Repair (Section 3), report the early results achieved so far (Section 4), and finally discuss future work (Section 5).

2 RELATED WORK

There are two main classes of approaches to automatic program repair: generate-and-validate and semantics-driven techniques.

Approaches based on *Generate-and-Validate* typically exploit Spectrum-Based Fault Localization (SBFL) [1, 22] to identify the set of suspicious statements in a faulty program and then iteratively modify these statements with a set of change operators to possibly obtain a plausible fix. Multiple change operators are normally exploited in a repair process, with the operator that copies statements or expressions from a program location to another one being the most popular operator used to achieve non-trivial changes (this operator assumes that a plausible fix can be obtained from the code already available in a program, a.k.a. the plastic surgery hypothesis [2]). These operators can be applied systematically [21], randomly [19], or in a genetic programming process [6] to generate many different program variants until either a *plausible solution* is found or the time budget expires. A plausible solution is a program variant that passes all the available test cases, which does not necessarily mean that it implements a correct fix. Indeed, a program variant that passes the available test cases is not guaranteed to be correct or acceptable by the developers, this is why the generated fixes are normally manually inspected before they can be finalized.

Approaches based on *semantics-driven* techniques also exploit classic or modified SBFL techniques [4] to identify the suspicious statements that must be targeted with the repair process. The behavior of the program in the suspicious locations is then analyzed to infer a specification of how the fixed statements should affect the executions to obtain a plausible fix. This information is exploited to define a program synthesis task whose solution is a piece of code that can replace the existing code to obtain the fix [8, 14, 15, 17].

Failure-Driven Program Repair defines ad-hoc repair tasks flexibly exploiting Generate-and-Valide and Semantics-Driven techniques, depending on the observed failure and the guessed fault that must be tentatively repaired. Moreover, Failure-Driven Program Repair does not use standard versions of these techniques, but customize them based on the specific needs of the faulty program, possibly obtaining extremely efficient repair processes.

Conceptually, Failure-Driven Program Repair might be considered close to Semantics-Driven techniques because they both exploit semantic information. However, the specific repair strategies are extremely diverse, in fact Failure-Driven Program Repair implements the repair process as a set of dedicated repair tasks that address the possible faults with semantics-driven techniques being just one of the options for the definition of a repair task.

3 FAILURE-DRIVEN PROGRAM REPAIR

The Failure-Driven Program Repair approach consists of the following three steps: (1) test execution, (2) failure-driven hypothesis formulation, and (3) failure-driven task generation. Figure 1 visually illustrates these steps.

The *Test Execution* step executes the available test suite on a faulty program in order to extract information about the failure. For instance, if a failing test case terminates with an exception, its stack trace usually contains relevant details about the failure, such as the failure type, the statement that threw the exception, and the set of pending method calls at the time of the failure.

The *Failure-Driven Hypothesis Formulation* step exploits this information, in addition to prior knowledge about the types of failures that can be addressed, to identify a set of repair targets, which can be effectively addressed with ad-hoc repair strategies. A repair target includes information about a guessed fault and its possible location, and can be derived directly from an observed failure. For instance, a failure originated by an `ArrayIndexOutOfBoundsException` may produce the following repair targets: the assignment that defines the value of the variable used as array index in the statement that produces the exception, assuming the index value is incorrect; the name of the array variable used in the statement that produces the exception, assuming the wrong array has been used; the statements just before the one that raises the exception, assuming the values of the variables involved in the exception have not been properly modified at least in the failing scenario. Note that the location of a repair target is often defined at a granularity level finer than statements (e.g., a variable in a statement rather than the full statement).

The repair targets are then managed by the *Failure-Driven Task Generation* step, whose aim is to define repair tasks that include specific repair strategies and optimal configurations to address the available repair targets. This step also organizes the repair tasks according to a suitable scheduling policy.

The concrete *repair tasks* associated with the repair targets are the results of two levels of specialization. The higher level of specialization consists of the choice of the *repair strategy*, while the fine-grained level of specialization is the configuration of the selected repair strategy. For the moment, I envision the usage of three possible strategies: systematic, genetic, and synthesis-based.

The systematic strategy consists of systematically applying simple change operators to the target locations until exhausting the possible changes that can be produced [21]. The configuration consists in the selection of change operators designed ad-hoc to address the guessed fault. For instance, if the guess is that a wrong array variable has been used in a program location, the systematic strategy can be exploited to systematically change the array variable with every other array variable defined in the scope.

The genetic strategy exploits genetic programming to generate and explore non-trivial fix spaces [6]. However, compared to the general problem of using genetic strategies to explore huge spaces with very few plausible fixes [11], the failure-driven approach defines a very limited and well focused fix space that must be explored. I achieve this by limiting the application of the approach to a specific

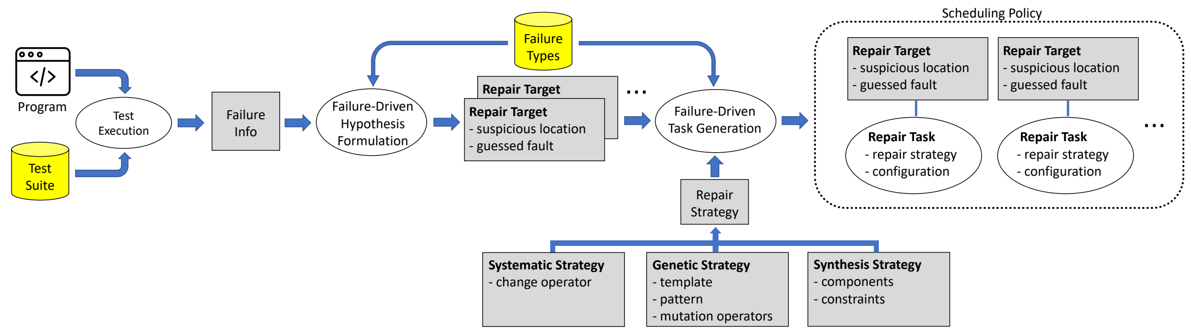

Figure 1: Failure-Driven Program Repair.

program location, possibly changed according to a template that captures the guessed high-level structure of the fix. For instance, the template might consist of the addition of a new `if` condition with its body, and the genetic strategy is used to explore possible variants for the condition and the body. In addition, the genetic strategy is customized in terms of the mutation operators that must be used to modify the code and the ingredients that the mutation operators can exploit in the repair task. For example, the ingredients may consist of conditions and assignments extracted from the faulty program and that can be used to define the `if` condition and its body, while the mutation operators may alter variables, constants and expressions to consider additional fixes.

The synthesis strategy can generate code fragments based on a set of constraints formulated by analyzing the program during the execution of the test cases [17]. Failure-Driven Task Generation exploits program synthesis in a specialized form to produce fragments that may fix the guessed faults. For instance, if the guessed fault is a faulty initialization of an array variable, program synthesis can be used to produce a new expression that combines components that are typically used in array initializations, such as integer variables, constants, arithmetic operators, and the length attribute of arrays.

Failure-Driven Program Repair generates multiple repair tasks that may exploit different repair strategies to address the repair targets generated for a same problem. The choice depends on the strategy that is assumed to be more effective for each guessed fault. For instance, changing variable names can be done systematically, while generating new code fragments may require strategies based on synthesis or genetic algorithms depending on the complexity of the fix that should be generated.

These tasks are processed according to a scheduling policy that also depends on the deployment of the program repair framework. For example, repair targets might be processed one after the other with a timeout assigned with each repair task. Alternatively, all the repair tasks can be worked out in parallel, or in pools.

A key aspect to stress about the program repair framework investigated in this Ph.D. thesis is that it is entirely failure-driven and it extensively exploits both the semantic of failures and the guessed faults to define highly specialized and effective repair tasks. That is, unlike the other approaches, Failure-Driven Program Repair dynamically adapts itself to the specific type of failure that is addressed. Moreover, the approach can potentially handle more and more program faults and contexts by increasing the knowledge base of the failure types and the set of strategies and configuration options available.

4 VALIDATION OF THE WORK

I am currently developing Failure-Driven Program Repair as an extension of ASTOR [13], which is a general-purpose program repair framework. To evaluate the approach, I focused on the faults contained in the DEFECTS4J benchmark [7], which has been used as benchmark by several other program repair techniques.

Among the faults contained in the DEFECTS4J benchmark, there are faults that are outside the capabilities of existing program repair techniques. To discuss the capabilities of Failure-Driven Program Repair, here I discuss how the approach can address a fault that cannot be repaired with other approaches (Project Math, Bug ID 3). The considered fault causes one of the test cases in the program test suite to fail throwing an `ArrayIndexOutOfBoundsException`. The fix requires adding to the program a fairly complicated new code block that is

```
if (len == 1) {
    return a[0] * b[0];
}
```

Failure-Driven Program Repair addresses this fault by first extracting relevant information about the failure. In addition to the type of exception raised, it extracts the location accessed in the array, and the statement that raised the exception. Based on this information, it formulates a number of hypotheses about possible faults and their locations, considering the elements involved in an access to an array, generating the following repair targets:

(1) The array variable is wrong and it is necessary to replace it with another variable of the same type;
(2) The index used to access the array is wrong and it is necessary to replace it with another constant or integer expression;
(3) The size of the array is wrong and it is necessary to change its initialization;
(4) The variables involved in the statement are correct but it misses a piece of logic that handles the case that led to the exception.

For each repair target, Failure-Driven Program Repair generates a repair task. The first two repair targets are addressed by instantiating the systematic repair strategy configured to work with variable replacement operators that only try with the relevant variables in scope. The third repair target is addressed with the genetic strategy specifically instantiated to alter the initialization statement (note that this program location is different from the location that generates the exception). Finally, the last repair target is addressed selecting the program location preceding the point that raises the exception and adding a new `if` code block with its body that is altered with the genetic strategy. In particular, the conditions and the statements in the body are extracted from the program and then altered with mutation operators.

Note that the first two repair tasks can quickly fix simple faults (e.g., a variable that must be replaced), while the third and the fourth tasks address more complex scenarios. In this example, the last strategy is the one that fixes the fault. Interestingly it first creates the code

```
if (len == 0) {
    return prodLowSum;
}
```

that is then mutated until achieving the following implementation

```
if (len == 1) {
    return prodHighCur;
}
```

that is a fix equivalent to the one implemented by the developers since the value of `prodHighCur` is exactly `a[0] * b[0]`. Note that adding an entire new `if` condition with a return statement in the body is practically prohibitive for other program repair approaches.

So far, the initial results concern with a number of specific cases, but I plan to quickly complete the implementation and evaluate the approach with benchmarks, such as DEFECTS4J [7] and BEARS [12], and new faults.

5 FUTURE WORK

In the initial phase of my Ph.D. work I focused on the definition of the Failure-Driven Program Repair process, the definition of repair strategies based on failures that produce exceptions, and the implementation of the approach based on systematic and genetic repair strategies. The initial results demonstrated that the approach can perspectively repair faults that cannot be addressed with state of the art approaches, as discussed in Section 4.

In the next phase, I plan to complete the prototype implementation of the approach and experiment it with both standard benchmarks available in the field [7, 12], and new sets of faults. The prototype will be released as an open source artefact. This effort will result in the first evidence of the effectiveness of Failure-Driven Program Repair released to the community.

In the last phase of my Ph.D. thesis I plan to incrementally extend the applicability of the approach by gradually supporting new types of exceptions and other types of failures not raising exceptions, and by including synthesis techniques among the supported repair strategies. Also in this case, I plan to release a reference implementation and extensive evidence of its effectiveness. In this context, the impact of the scheduling policy on the results will also be considered.

ACKNOWLEDGMENTS

The author is advised by Prof. Leonardo Mariani. This work has been partially supported by the EU H2020 "Learn" project, which has been funded under the ERC Consolidator Grant 2014 program (Grant Agreement n. 646867).

REFERENCES

[1] P. Agarwal and A.P. Agrawal. 2014. Fault-localization Techniques for Software Systems: A Literature Review. *SIGSOFT Software Engineering Notes* 39, 5 (2014), 1–8. doi:10.1145/2659118.2659125.
[2] E. T. Barr, T. Brun, P. Devanbu, M. Harman, and F. Sarro. 2014. The plastic surgery hypothesis. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*. 306–317. doi: 10.1145/2635868.2635898.
[3] T. Britton, L. Jeng, G. Carver, and P. Cheak. 2013. Reversible Debugging Software - Quantify the time and cost saved using reversible debuggers.
[4] F. DeMarco, J. Xuan, D. Le Berre, and M. Monperrus. 2014. Automatic repair of buggy if conditions and missing preconditions with SMT. In *Proceedings of the International Workshop on Constraints in Software Testing, Verification, and Analysis (CSTVA)*. 30–39. doi: 10.1145/2593735.2593740.
[5] L. Gazzola, D. Micucci, and L. Mariani. 2019. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering (TSE)* 45, 1 (2019), 34–67.
[6] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. 2012. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering (TSE)* 38, 1 (2012), 54–72. doi: 10.1109/TSE.2011.104.
[7] R. Just, D. Jalali, and M. D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. ACM, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055
[8] Y. Ke, K. Stolee, C. Le Goues, and Y. Brun. 2015. Repairing Programs with Semantic Code Search. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. 295–306. doi: 10.1109/ASE.2015.60.
[9] X.-B. D. Le, D. Lo, and C. Le Goues. 2016. History driven program repair. In *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 213–224. doi: 10.1109/SANER.2016.76.
[10] F. Long and M. Rinard. 2016. Automatic patch generation by learning correct code. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 298–312.
[11] F. Long and M. C. Rinard. 2016. An analysis of the search spaces for generate and validate patch generation systems. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 702–713.
[12] F. Madeiral, S. Urli, M. Maia, and M. Monperrus. 2019. Bears: An Extensible Java Bug Benchmark for Automatic Program Repair Studies. In *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '19)*. https://arxiv.org/abs/1901.06024
[13] M. Martinez and M. Monperrus. 2016. ASTOR: A Program Repair Library for Java. In *Proceedings of ISSTA*. https://doi.org/10.1145/2931037.2948705
[14] S. Mechtaev, J. Yi, and A. Roychoudhury. 2015. Directfix: Looking for simple program repairs. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 448–458. doi: 10.1109/ICSE.2015.63.
[15] S. Mechtaev, J. Yi, and A. Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 691–701. doi: 10.1145/2884781.2884807.
[16] M. Monperrus. 2018. Automatic Software Repair: A Bibliography. *ACM Computing Survey* 51, 1, Article 17 (Jan. 2018), 24 pages.
[17] H. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. 2013. SemFix: Program repair via semantic analysis. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 772–781. doi:10.1109/ICSE.2013.6606623.
[18] C. Parnin and A. Orso. 2011. Are automated debugging techniques actually helping programmers?. In *Proceedings of the 2011 international symposium on software testing and analysis*. ACM, 199–209.
[19] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. 2014. The strength of random search on automated program repair. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 254–265. doi: 10.1145/2568225.2568254.
[20] Undo Software. 2014. *Increasing software development productivity with reversible debugging*. Technical Report white paper. Undo Software.
[21] W. Weimer, Z. Fry, and S. Forrest. 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. 356–366. doi: 10.1109/ASE.2013.6693094.
[22] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.
[23] 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 TSE* 43, 1 (2017), 34–55. doi: 10.1109/TSE.2016.2560811.
[24] A. Zeller and R. Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200.