Two Flavors in Automated Software Repair: Rigid Repair and Plastic Repair

Martin Monperrus, Benoit Baudry

Dagstuhl Preprint, Seminar #13061, 2013.
Link to the latest version

Abstract

In this paper, we discuss two families of automated software repair approaches that we call "rigid repair" and "plastic repair". We shape the notions of rigid repair and plastic repair around the perception of software correctness. Rigid repair relies on a binary notion of "bug" and "repair". Plastic repair refers to the plasticity of software, both in terms of correctness and in terms of intrinsic characteristics.

1 Introduction

Automated software repair consists of automatically fixing buggy programs, for instance by generating a patch. This research field is related to fault localization (understanding the cause and the failure chain leading to a bug) and code synthesis (generating code from a declarative specification).

In this paper, we discuss two families of automated software repair approaches that we call "rigid repair" and "plastic repair". If the software engineering community finds a consensus on those names and concepts, it will enable authors to more easily place their work in the right context and the audience to judge new repair techniques with the correct mindset.

We shape the notions of rigid repair and plastic repair around the perception of software correctness. The software correctness results from the decision process that distinguishes a piece of software as buggy or not (from the viewpoint of users or engineers).

2 Software Correctness and Repair

2.1 Dimensions of Correctness

The conventional, common sense, notion of software correctness is binary: there is a test procedure that says whether the software is correct (the procedure outputs "true") or not (the procedure outputs "false"). Boolean assertions in programming languages and testing frameworks embody this notion.

Binary correctness is “rigid” and defines a rigid correctness envelop: inside the envelop the software is correct, outside, it’s not.

Software correctness exists at different scales: e.g. at the level of expressions (e.g. arithmetic expressions), functions, modules, systems, etc. Small-scale correctness is often binary. However, when the scale increases, a different kind of correctness appears from the combination of locally binary correct elements.

Let us consider a system with 2000 test cases. A system “A” that passes 1900 test cases is more correct than one that passes 1800 test cases. From the binary correctness emerges an ordering (“B is less correct than A”). If the system manipulates continuous values, the correctness ordering can even be continuous (that holds for small-scale correctness as well). Furthermore a system that only passes the first 1000 tests is not comparable with a system that only passes with the remaining 1000 ones. From the binary correctness emerges incommensurability (“A and B are not comparable”).

In other terms, we think that it is wrong is to go from "local binary correctness" to "global binary correctness" by induction. Software correctness can be locally binary but globally continuous and partial. We call this global continuous partial correctness, “plastic correctness”. Compared to the rigid correctness envelop, the plastic correctness envelop deals with uncertainty.

Plastic correctness envelop fits the class of software systems for which it is not possible to have a binary oracle, e.g., systems that use heuristics to compute certain decisions, disseminate information on networks or manage resources. To our knowledge, this plastic correctness has been called “acceptability envelope” [12], “controlled uncertainty” [6] and “approximate computation” [10].

To sum, different notions of software correctness yield different correctness envelops. There is an intimate link between correctness envelop and repair. The knowledge of the correctness envelop can drive the automated repair process, the sharpness of the correctness boundary enables one to say that a fix is fine or not with more or less certainty.

2.2 Rigid Repair

Rigid repair approaches rely on a rigid, binary definition of software correctness, whether local or global. The literature is rich of various formalisms to specify this envelop, such as contracts [8] and typestates [3]

Having this binary explicit correctness envelop is very useful for repair. Rigid repair techniques consists of bringing the software back into the correctness envelope. The rigid repair is guided by the knowledge of the envelope itself, the envelop driving the synthesis of bug fixes. We use the term rigid to convey the idea of being able to touch the envelop and to some extent to “climb” on it. For instance, Wei et al. uses the contracts of Eiffel programs (pre- and post-conditions) to drive the repair process [16]. Nguyen et al.

uses symbolic execution of C programs to fix arithmetic and initialization bugs [11].

2.3 Plastic Repair

Plastic repair approaches refer to the plasticity of software. In the context of automated software repair, by “plasticity”, we mean two things.

First, plastic repair refers to plastic software correctness. For instance, automatic input rectification [7] and loop perforation [9] are two kinds of repair approaches that work on top of plastic correctness.

Second, it seems that in a certain problem domains and technology, software is “plastic” in the biological sense, as DNA is plastic: it can be reused partly and in unanticipated ways [15], it is intrinsically redundant [4] and supports mutations [14]. Previous work has shown that this plasticity can be leveraged for software repair. For instance, Carzaniga et al. [1] uses the redundancy for recovering from runtime failures.

3 Discussion

To sum up, rigid repair relies on rigid structures to drive the automated repair process, in particular the structure of the correctness envelop. Plastic repair uses plastic concepts either at the level of the correctness envelope or at the level of the code being manipulated.

This has an impact on the techniques being involved in the repair process, rigid repair tends to use automated reasoning (e.g. [11]) and plastic repair techniques are well appropriate to randomized algorithms (e.g. [2]). Consequently, rigid repair tend to be deterministic and bounded in time while plastic repair does not. Note that even for rigid repair, execution time can be in practice infinite in non-polynomial settings.

It seems that both families of repair techniques can be formalized. For instance, Samimi et al. [13] presented a rigid approach that is complete: if there is a fix, it will be found. Zhu et al. [17] showed that loop perforation (very plastic) can be achieved within strong accuracy bounds.

The distinction between rigid and plastic repair is not exclusive. For instance, Le Goues et al. [5] use a plastic characteristic of code – its redundancy – on top of a binary correctness envelope (specified by a test suite). We would call it “plastic” because its usage of code redundancy (a kind of code plasticity) and its abundant exploration of the unspecified correctness domain (by manipulating code parts that are not covered by test cases or input data [14]).

To conclude, automated software repair is an active field, and new results will surely contribute to precise the concepts of plastic repair and rigid repair. Many questions are open: Which classes of bugs are not amenable to being specified by a binary correctness envelop? How to characterize and recognize plastic problem domains as well as zones of plasticity in source code? Is there a relation between the plasticity of correctness and the plasticity of code itself?

References

[1] A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino, and M. Pezzè. Automatic recovery from runtime failures. In *Proceedings of ICSE'13*, 2013.

[2] V. Debroy and W. Wong. Using mutation to automatically suggest fixes for faulty programs. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 65–74. IEEE, 2010.

[3] R. DeLine and M. Fähndrich. Typestates for objects. In *Proceedings of ECOOP*, volume 3086, pages 465–490, June 2004.

[4] M. Gabel and Z. Su. A study of the uniqueness of source code. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 147–156. ACM, 2010.

[5] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38:54–72, 2012.

[6] M. E. Locasto, S. Sidiroglou, and A. D. Keromytis. Software self-healing using collaborative application communities. In *NDSS*, 2006.

[7] F. Long, V. Ganesh, M. Carbin, S. Sidiroglou, , and M. Rinard. Automatic input rectification. In *Proceedings of ICSE*, 2012.

[8] B. Meyer. Applying design by contract. *Computer*, 25(10):40–51, 1992.

[9] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. C. Rinard. Quality of service profiling. In *ICSE (1)*, pages 25–34, 2010.

[10] S. Natarajan. *Imprecise and Approximate Computation*, volume 318. Springer, 1995.

[11] H. D. T. Nguyen, D. Qi, A. Roychoudhury, , and S. Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of ICSE'13*, 2013.

[12] M. Rinard, C. Cadar, and H. H. Nguyen. Exploring the acceptability envelope. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 21–30. ACM, 2005.

[13] H. Samimi, M. Schäfer, S. Artzi, T. D. Millstein, F. Tip, and L. J. Hendren. Automated repair of html generation errors in php applications using string constraint solving. In *ICSE*, pages 277–287, 2012.

[14] E. Schulte, Z. P. Fry, E. Fast, W. Weimer, and S. Forrest. Software mutational robustness. *arXiv preprint arXiv:1204.4224*, 2012.

[15] S. Thummalapenta and T. Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 204–213. ACM, 2007.

[16] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *Proceedings of the International Symposium on Software Testing and Analysis*. AC, 2010.

[17] Z. A. Zhu, S. Misailovic, J. A. Kelner, and M. C. Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. In *POPL*, pages 441–454, 2012.