BUGFIX: towards a common language and framework for the Automatic Program Repair community

Bertrand Meyer
Viktoryia Kananchuk
Li Huang
Bertrand.Meyer@inf.ethz.ch
viktoryia.kononchuk@sit.study
Li.Huang@constructor.org
Constructor Institute
Schaffhausen, Switzerland

ABSTRACT

Techniques of Automatic Program Repair (APR) have the potential of thoroughly facilitating the task of producing quality software. After a promising start, however, progress in making APR practical has been hindered by the lack of a common framework to support the multiplicity of APR ideas and tools, and of target programming languages and environments. In this position paper we outline a general framework to enable the APR community to benefit from each other's advances, in particular through a standard language for describing bugs and their fixes. Such a common framework — which is also applicable to work on fault seeding — could be a tremendous benefit to researchers and developers of Interactive Development Environments (IDEs) who are working to make APR an effective part of the software developer's practical experience.

CCS CONCEPTS

• **Software and its engineering → Formal software verification**; **Software testing and debugging**; **Empirical software validation**; **Error handling and recovery**.

KEYWORDS

Automatic Program Repair, Debugging, Integrated Development Environments, Software tools, Program transformation, Bug seeding, Software quality

ACM Reference Format:
Bertrand Meyer, Viktoryia Kananchuk, and Li Huang. 2024. BUGFIX: towards a common language and framework for the Automatic Program Repair community. In *2024 ACM/IEEE International Workshop on Automated Program Repair (APR '24), April 20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3643788.3648007

APR '24 , April 20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0577-9/24/04…$15.00
https://doi.org/10.1145/3643788.3648007

1 INTRODUCTION

We expect software to work properly; failures to meet that expectation, known as faults or bugs, can have serious consequences, all the more serious that software is now pervasive in the pursuit of almost all human affairs. Yet, just as the best-behaved children will occasionally do something silly and the most careful drivers will occasionally get a speeding ticket, programmers will occasionally produce buggy software. Not just human programmers but automated ones as well: Artificial Intelligence assistants such as ChatGPT and Copilot, while impressive in their capabilities, cannot be trusted to produce bug-free software; they mess up just as much as we, mere mortals [12].

Techniques of software verification, from tests to static analysis and proofs, help identify bugs; in recent years the idea has emerged that while spotting a bug is good, correcting the bug — or, more realistically, suggesting a correction to the programmer — is better. Some early work in this direction includes [9, 15–17, 20, 23]; good surveys can be found in [6, 13, 14]. Ideally, Automatic Program Repair (APR) should be included in the basic toolset that programmers use (IDE, Interactive Development Environment), so that if a programmer produces a potentially faulty code element, the IDE's verification tools silently detect it and pop up a message that both signals the bug and suggests one or more valid corrections. Making this scenario possible is the common goal — the common Graal — of all developers and researchers working on APR. One of the major obstacles to reaching it is the heterogeneity of the field. Heterogeneity of APR approaches; diversity of IDEs; diversity of programming languages; and also diversity of bugs. These heterogeneity factors force every APR project to invest a major part of its effort in recreating a basic bug and fix description framework; such spurious repetition of work considerably impair progress towards making Automatic Program Repair a standard and effective part of the software developer's daily experience. The goal of the work-in-progress described here is to overcome these obstacles by establishing a joint framework, BUGFIX, that all APR efforts can use. The present paper is explicitly intended as a workshop position paper, intended for discussion and feedback rather than presenting a fully developed solution. Our intent is to present BUGFIX at the workshop to elicit feedback from the APR community and go on to develop a finalized version with the best chances of being widely adopted, and as a result speeding up the development of Automatic Program Repair.

```
syntax CALL for Java:
    r (args)
syntax CALL for Eiffel:
    [args.count ≠ 0 → r (args) | r]
```

2 A BUG-AND-FIX SPECIFICATION LANGUAGE

At the core of BUGFIX lies a language for describing identified patterns for both bugs and their fixes. Two observations are in order:

- The BUGFIX effort focuses on bugs that manifest themselves through code patterns that are wrong and should be replaced (fixed) by adapting the patterns. For example, code that uses `f (a, b)` when it should use `f (b, a)`. We do not at this point consider deeper or more elaborate bugs, such as design bugs, as analyzed for example in [2].
- To illustrate the BUGFIX language, we use a concrete syntax. For clarity the syntax is keyword-oriented and Eiffel-like. Concrete syntax details are not important, however, for the concepts discussed here; the syntax may change in the future. What matters is the abstract syntax and underlying semantic concepts. In many practical applications we expect that BUGFIX will be used not through a human-oriented syntax such as the one illustrated below but through a program-oriented API (Abstract Program Interface), for use by APR tools and databases.

The following example, using an ad hoc concrete syntax as just mentioned, illustrates the argument-reversal bug and its fix.

BUGFIX should support a wide range of programming languages. It needs, however, to provide bug and fix descriptions for language mechanisms, or "constructs", that exist (in different forms) in various languages. Examples include routine call, assignment, loop and so on. The specification of language constructs is the first part of BUGFIX.

The mechanism supports both the description of a general pattern for each construct and the specification of its realization in a particular language; the latter should be extendible, so that one can add new languages and their implementation of predefined general constructs. The general specification of "routine call" could correspondingly appear as

```
construct CALL feature
    args: EXPRESSION*
    r: ROUTINE
end
```

where the constructs `EXPRESSION` and `ROUTINE` are separately defined. The specification of `CALL` simply state that a routine call includes a routine name (`r`) and a list of actual arguments (`args`), each of which is an instance of `EXPRESSION`. The specific specifications syntax (in Java and Eiffel) are below:

Note the conditional expression in the Eiffel case: a call with actual arguments uses parentheses, as in "`r (a, b, c)`", but a call without arguments is (unlike in Java) just "`r`" without parentheses. BUGFIX includes support for such conditional mechanisms.

The separation between a general language construct, such as `CALL`, and the specification of its (concrete) syntax for any particular programming language, contributes to the generality of BUGFIX, its usability by many different APR projects targeting at different languages and IDEs, and its extendibility to new environments.

Once the language constructs have been identified, we may proceed to the core goal of BUGFIX — specifying bug-and-fix patterns. Here is the swapped-argument example:

```
pattern SWAPPED_ARGUMENTS for
    c: CALL
with
    a1, a2: EXPRESSION
where
    a1 ∈ c.args ; a2 ∈ c.args
    a1.index ≠ a2.index
fix
    c [a1 ← a2, a2 ← old a1]
end
```

The intent should be clear: we describe a program transformation whereby two out-of-order arguments are swapped.

The following example covers two commonly encountered bugs: a programmer uses a sum instead of a difference (`PLUS_MINUS`), or use of equality operator instead of inequality (`EQ_NEQ`).

```
pattern PLUS_MINUS for
    e: SUM
with
    e1, e2: EXPRESSION
where
    e1 = e.first
    e2 = e.second
fix
    DIFFERENCE [first ← e1, second ← e2]
end
```

```
pattern EQ_NEQ for
    e: EQ_BIN_OP
with
    e1, e2: EXPRESSION
where
    e1 = e.first
    e2 = e.second
fix
    NEQ_BIN_OP [first ← e1, second ← e2]
end
```

The examples also illustrate the purpose — and limits — of the BUGFIX language. Two arguments may or may not be in the right order; swapping them may or may not be the proper fix. Deciding on these questions is beyond the scope of BUGFIX: it is the task of Automatic Program Repair methods and tools. BUGFIX does not attempt to provide a magical wand for APR, but provides the (brilliant) devisers of magic wands with a way to develop, test, validate, experiment, refine, explain, implement and publicize their

contributions, and to compare their success in APR, in objective ways, to the results of magic wands produced by other (brilliant) developers.

3 GENERALITY ASSESSMENT

The above examples are simple, but the current initial version of BUGFIX, using the ideas just outlined through these examples, make it possible to cover a wide range of common bugs.

To address the issue of generality, our design goal for BUGFIX has not been to address the widest possible range of conceivable bugs, which might lead to an ambitious but unusable contraption; we have taken instead the pragmatic approach of looking at the most common types of bugs; specifically, bugs that are both:

- Actual code bugs (rather than high-level design or requirements bugs which, as noted, are harder to handle), with clear applicable fixes.
- Often found in actual code, as illustrated — objectively — by the empirical study of large software repositories (Linux, Eclipse, Apache).

We call such bugs, the ones most conducive to successful APR work, **Low-Hanging Bugs** (LHBs). Fortunately a significant amount of work already exists on the analysis of bugs and fixes for major software repositories [7]. While not complete, it provides a good initial catalog of bugs, from which we started with our own ongoing studies of repositories and used them to collect LHBs.

This analysis of bugs through both the literature and our own studies leads to a set of Low-Hanging Bugs that seems to occur widely. Specifically, the current analysis uses the Defects4J [7] dataset, which is a widely used benchmark for automatic repair of Java programs. Through an analysis of the 364 representative bugs in the Defects4J dataset, we find 51 LHBs, accounting for 14% of the total. For the rest of the non-LHBs (86%), their fixes either involve multiple lines of code or cover various types of program constructs, which makes it difficult to generalize fix patterns. These ever-recurring bugs, identified throughout repositories, appear worthy of particular attention for APR research. The most significant sub-categories are:

- Missing Null checking (25.5% of LHBs)
- Incorrect variables (23.5%)
- Bugs related to -/+1 (13.7%)
- Misuse of order operators (<, ≤, ≥, > etc.) (11.7%)
- Misuse of False/True (7.8%)

The full list of categories and the current result of our analyses on Defects4J is available in the Github repository[1]. As ongoing work, we also aim to include the EiffelBase library [11] in the analysis — the initial implementation of EiffelBase had a significant number of bugs, many of them non-trivial, and has served as the basis for several earlier studies of bugs and fixes [16, 17, 21]. We intend to continue to maintain and develop this repository as a general community resource to help APR researchers and product developers.

The potential for developing the BUGFIX language for generality is in principle unbounded: its scope could theoretically encompass any program or design transformation. For BUGFIX to be of practical use, it should strike the right balance between generality and simplicity. We are adopting a cautious approach favoring the second of these criteria, keeping BUGFIX a small language, guided by its applicability to bug and fixes that do appear frequently in practice — in other words, Low-Hanging-Bugs, where the inspiration comes not from our own intuition or opinions but from the empirical, objective study of credible bug and fix repositories.

[1]BUGFIX: github.com/apr-2024/BUGFIX

4 ABSTRACT INTERFACE

To guide the specification and further development of BUGFIX, we are relying both on an example concrete syntax illustrated above but, more fundamentally, on an abstract programming interface (API). The original version of the interface is written in Eiffel, since this language offers powerful abstraction mechanisms, particularly the Design by Contract specification facilities; for example the "`where`" clause of BUGFIX illustrated by the `PLUS_MINUS` example above can directly be expressed by an Eiffel "precondition" (`require` clause). From this basic form of expressing BUGFIX elements (constructs and bug-fix patterns), we will make available others, notably:

- The concrete syntax form, as illustrated, for easy human comprehension.
- Libraries in other common languages, such as Java or Python (using an open mechanism allowing community contributions).
- Non-programming-language forms, for direct consumption by tools, in binary formats or exchange formats such as JSON or XML.

The aim is to encourage the development of a wide range of bug and bug-fix patterns, open to contributions by all members of the APR community.

5 FURTHER APPLICATIONS

BUGFIX supports the specification of language constructs and bug-fix patterns. These patterns are, more generally, code-transformation schemes. Besides their application to Automatic Program Repair, which the main focus of this article, they have potential uses for other efforts involving predefined schemes for program transformation. Two notable potential examples are:

- **Fault seeding**. A common technique in testing, and more specifically the analysis of test case quality, is to introduce faults (bugs) artificially into programs. For the validity of the corresponding applications, these faults should as much as possible reflect the patterns of actual bugs produced by programmers. To describe fault-seeding schemes, one can use BUGFIX: the fix part describes the correct pattern, and the bug part describes the seeded bug.
- **Verification condition generation**. Software verification, for example in Hoare-style axiomatic semantics, needs assertions about the program, for example loop invariants. A modern, effective software verification environment should help programmers produce such "verification conditions", facilitating a task which can be conceptually hard and, even when it is not, remains tedious and time-consuming. The Daikon tool [5] has pioneered this line of research by proposing invariant patterns. Such patterns can be described in BUGFIX (ignoring in this case the "fix" part of specifications).

More generally, program transformation is a recurring need of software development tools, arising in many different applications; BUGFIX can provide a general program-transformation specification framework.

6 RELATED WORK

The idea of identifying and/or analyzing common bug or fix patterns is not new, but it has been mostly applied in specific contexts or programming languages: the analysis of bug and fix patterns [1, 18] focuses on the patterns for Java program; the work presented in [10] and [22] identified project-specific bug patterns with focuses on method call pairs (two methods should be called in pairs) or checking return value. Sun et al. [19] identified different categories of bugs and commonly used fix patterns in Machine Learning projects. In contrast to the above work, the approach presented in this paper aims to provide a unified framework that allows to derive bug and fix patterns in a more general context, involving various languages or application domains.

Duraes et al. [4] presented a classification of defects, which is an extension of the Orthogonal Defect Classification (ODC) [3], with the main focus on the Emulation of Software Faults. Catolino et al. [2] presented an taxonomy of bug root causes. The primary motivation of their work, however, lies in expediting error resolution for developers, not identifying commonly used bug and fix patterns.

A repository described in [8] contains a same set of bugs in two different languages (Java and Python); it provides a benchmark for evaluation of multilingual APR tools. BUGFIX shares the similar vision – to serve as a common platform that allows evaluation and comparison of APR techniques involving different programming languages.

7 LIMITATIONS, CONCLUSIONS AND FUTURE WORK

This workshop contribution describes the inception and first results of a project intended to provide the Automatic Program Repair with a common frame of reference, BUGFIX, including a conceptual framework, a language in the complete sense of the term (abstract syntax, API, concrete syntax, hooks to an extendible set of programming languages), and a rich and reliable repository of important and representative examples of bugs and possible fixes, allowing them (as noted at the end of section 2) to apply their insights and artifacts to standard examples and validate their effectiveness against other work.

The limitations of the current state of the work have been made clear throughout this article. The most obvious is its work-in-progress nature. It is important, however, to note that the key design decisions have been made (as described in the preceding sections) and are here to stay. They follow from a careful examination of key criteria and an attempt at obtaining an effective tradeoff; the criteria are: simplicity; learnability; generality; practicality (based on a long-running analysis of actual bugs as actual people produce them, from the empirical analysis of actual software repositories). They determine the essential nature of the BUGFIX framework and are the principal contribution of the work.

The self-admitted tentative nature of the present description is not just, however, an item in its list of limitations. It is also, more importantly, one of its intentional features. This article is a submission to a workshop and has been designed that way. Instead of a solution claiming to be fully finalized, which might fit the authors' personal Weltanschauung of software engineering but fail to miss the real needs of many developers, it proposes an initial version of BUGFIX, sufficiently explicit and developed to show the essential insights, contributions and applications, but still open to refinement. By using this approach, we expect to elicit constructive feedback from the APR community and steer the future development of BUGFIX towards a final result that will be the most useful possible to a broad set of APR innovators.

Already in its current state as specified in the preceding sections, BUGFIX provides a general framework for describing and processing bugs and their fixes. We hope that its introduction will — even in a modest way — help advance one of the key goals towards the more general quest by the software engineering profession to provide the world with better software: the goal of equipping software developers — when they occasionally let bugs slip into their programs, as almost all of them with the possible exception of Donald Knuth inevitably do, and will continue to do for a long, long time — with effective, useful and correct suggestions of Automatic Program Repair.

REFERENCES

[1] Eduardo Cunha Campos and Marcelo de Almeida Maia. 2017. Common bug-fix patterns: A large-scale observational study. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 404–413.

[2] Gemma Catolino, Fabio Palomba, Andy Zaidman, and Filomena Ferrucci. 2019. Not all bugs are the same: Understanding, characterizing, and classifying bug types. *Journal of Systems and Software* 152 (2019), 165–181.

[3] Ram Chillarege, Inderpal S Bhandari, Jarir K Chaar, Michael J Halliday, Diane S Moebus, Bonnie K Ray, and Man-Yuen Wong. 1992. Orthogonal defect classification-a concept for in-process measurements. *IEEE Transactions on software Engineering* 18, 11 (1992), 943–956.

[4] Joao A Duraes and Henrique S Madeira. 2006. Emulation of software faults: A field data study and a practical approach. *Ieee transactions on software engineering* 32, 11 (2006), 849–867.

[5] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of computer programming* 69, 1-3 (2007), 35–45.

[6] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2018. Automatic software repair: A survey. In *Proceedings of the 40th International Conference on Software Engineering*. 1219–1219.

[7] René Just, Darioush Jalali, and Michael 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*. 437–440.

[8] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. QuixBugs: A multi-lingual program repair benchmark set based on the Quixey Challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*. 55–56.

[9] Kui Liu, Anil Koyuncu, Kisub Kim, Dongsun Kim, and Tegawendé F Bissyandé. 2018. LSRepair: Live search of fix ingredients for automated program repair. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 658–662.

[10] Benjamin Livshits and Thomas Zimmermann. 2005. Dynamine: finding common error patterns by mining software revision histories. *ACM SIGSOFT Software Engineering Notes* 30, 5 (2005), 296–305.

[11] Bertrand Meyer. 1994. *Reusable software: the Base object-oriented component libraries*. Prentice-Hall, Inc.

[12] Bertrand Meyer. 2023. AI Does Not Help Programmers. *Blog article at Communications of the ACM (CACM)* (3 June 2023).

[13] Martin Monperrus. 2018. Automatic software repair: A bibliography. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 1–24.

[14] Martin Monperrus. 2018. *The living review on automated program repair*. Ph. D. Dissertation. HAL Archives Ouvertes.

[15] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *International Conference on Software Engineering (ICSE)*. IEEE, 772–781.

[16] Yu Pei, Carlo A. Furia, Martin Nordio, and Bertrand Meyer. 2015. Automated Program Repair in an Integrated Development Environment. In *International Conference on Software Engineering (ICSE)*, Vol. 2. IEEE, 681–684.

[17] Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. 2014. Automated Fixing of Programs with Contracts. *Transactions on Software Engineering* 40, 5 (2014), 427–449.

[18] Mauricio Soto, Ferdian Thung, Chu-Pan Wong, Claire Le Goues, and David Lo. 2016. A deeper look into bug fixes: patterns, replacements, deletions, and additions. In *Proceedings of the 13th International Conference on Mining Software Repositories*. 512–515.

[19] Xiaobing Sun, Tianchi Zhou, Gengjie Li, Jiajun Hu, Hui Yang, and Bin Li. 2017. An empirical study on real bugs for machine learning programs. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 348–357.

[20] Rijnard van Tonder and Claire Le Goues. 2018. Static Automated Program Repair for Heap Properties. In *International Conference on Software Engineering (ICSE)*. ACM, 151–162.

[21] Yi Wei, Bertrand Meyer, and Manuel Oriol. 2012. Is branch coverage a good measure of testing effectiveness? *Empirical Software Engineering and Verification: International Summer Schools, LASER 2008-2010, Elba Island, Italy, Revised Tutorial Lectures* (2012), 194–212.

[22] Chadd C Williams and Jeffrey K Hollingsworth. 2005. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Transactions on Software Engineering* 31, 6 (2005), 466–480.

[23] Jifeng Xuan, Matias Martinez, Favio DeMarco, Maxime Clément, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *Transactions on Software Engineering* 43, 1 (2017), 34–55.