On the Introduction of Automatic Program Repair in Bloomberg

Serkan Kirbas, Etienne Windels, Olayori McBello, Kevin Kells, Matthew Pagano, Rafal Szalanski,
Bloomberg, London, UK & New York, USA

Vesna Nowack[1], **Emily Winter**[2], **Steve Counsell**[3], **David Bowes**[2], **Tracy Hall**[2], **Saemundur Haraldsson**[4], **John Woodward**[1]
[1]Queen Mary, University of London, UK
[2]Lancaster University, UK
[3]Brunel University, London, UK
[4]University of Stirling, UK

***Abstract*—A key to the success of Automatic Program Repair techniques is how easily they can be used in an industrial setting. In this article, we describe a collaboration by a team from four UK-based universities with Bloomberg (London) in implementing automatic, high-quality fixes to its code base. We explain the motivation for adopting APR, the mechanics of the prototype tool that was built, and the practicalities of integrating APR into existing systems.**

Introduction

As a way of assisting software engineers in their daily routine of fixing bugs, Automatic Program Repair (APR) is an area attracting increasing interest and involvement from industry [1], [2]. The benefits of the approach are clear. Why impose time-consuming and costly repair processes on software engineers when we can do it automatically [3]? In this article, we describe an initiative at Bloomberg's London offices exploring the incorporation of APR technology into its existing software development pipeline. A prototype tool, "Fixie", was developed and used by software engineers at Bloomberg for the past twelve months. The work represents joint research between the organisation and four UK-based universities, namely Lancaster, Brunel, Stirling and Queen Mary University of London. The university team have previously collaborated with one of the leads at Bloomberg [4] through a longstanding research programme at Brunel on bug analyses; two of the academic authors

 © 2020 IEEE

are currently on secondment as contractors at Bloomberg, helping to develop the APR tool and its integration into software engineers' workflow. Bloomberg is a formal, industrial partner on the research team and various project meetings have taken place both in the past and on an ongoing basis to facilitate the tool's development and to discuss progress and results.

Why Automatic Program Repair?

Over time, Bloomberg has observed a number of small, similar-looking bugs that occur often and yet always needed to be fixed manually in a similar way; this was taking up significant amounts of software engineer time in a repetitive, bug-fixing process. Bloomberg's vision in the short-term was not to provide a repair system for *all* of its bugs, but to deliberately target a more realistic and achievable set of small, repeatable bugs impacting its business. Addressing *only* those bugs would remove tedious and costly "grunt" work, free-up software engineers for more creative tasks and provide a catalogue of information which could help future development and project estimation processes.

The project faced a number of real-world constraints prior to APR introduction. Firstly, the initiative required management buy-in and a convincing cost-benefit analysis was needed by the Bloomberg APR team before the work could proceed; human resource limits and budgets were a key influencing factor. Secondly, software engineers (the Fixie users) were sceptical of the value of APR at the start and suspicious of the technology; change of any type can cause disruption, but asking software engineers to consider modifying their coding practices presented a problem. Thirdly, implementing APR required a range of technical skills and knowledge in transferring theory to practice and it took time and significant effort to develop the Fixie architecture. Finally, it was important that the APR approach conformed to existing coding and development standards at Bloomberg.

The Academic-Industry "Divide"

Generally speaking, adoption of APR by industry is still low. In terms of current industry initiatives, one prominent example is that of Facebook [5], [6]. The company has built a tool dedicated to the Facebook app called Getafix [1]. The tool is deployed in the app and finds fixes for bugs automatically; engineers then approve those fixes. Facebook claims that its tool both improves code quality and provides a more effective way of working for its engineers. However, any collaboration between academia and industry raises different perspectives and conflicting goals [7]. These were certainly brought to light in the case of collaboration between Bloomberg and the four universities involved in Fixie. These conflicts can be seen through multiple prisms which we now describe.

Fix Novelty

Academia often focuses on trying to find and then apply novel techniques for repairing bugs automatically which other academic-based APR tools have not a) repaired, or b) found solutions to before [8]. Put another way, academic APR research tends to focus on looking at the problem in a wider context and demonstrating that its approach is somehow better than all that have come before. Academia then sets about spending time solving those problems, often on a small subset of systems with limited transferable value. Indeed, academic research outputs often include statements suggesting that its approach finds fixes which cannot be found by any current repair techniques. While this aim is at the heart of what many would think was the purpose of academic research, Bloomberg is looking *only* for solutions to frequent and known bugs that were taking up inordinate amounts of its software engineers' time. Academics are often more interested in the novel aspects of research and finding the next new solution; industry, on the other hand, is more interested in the value they get from the technique, irrespective of whether it is novel or not.

Fix simplicity

Academia's view of APR is that it should strive to repair complex bugs automatically because this increases the applicability of an APR tool and demonstrates the wider viability of the approach [9]. Academic research will often claim that the work motivates new research for the improvement of APR application to important and hard to solve bugs. The reality in our case is that, counter-

intuitively, this makes an APR tool less applicable and less attractive to industry. On the other hand, Bloomberg was only interested in building a tool that could initially repair small, frequent and most importantly "trivial" bugs. Small fixes are easier to understand and explain to a software engineer and a complex repair does more to confuse a software engineer than to help, takes more time to understand and prevents its reuse. In summary, small and simple fixes were far more preferable to Bloomberg than complex fixes. If a small fix is populated throughout the system, it can save vast amounts of software engineering time and money. The smaller the repair, the better.

Fix verification

A commonly-held theme in academic research is that automatic verification of fixes is preferable to where there are elements of human intervention in the process [10]. Automated fix assessment is often stated as the objective of APR research. One explanation is that maybe academics always take a purist's point of view: APR should be an end-to-end automatic process and strive for complete automation without human intervention. However, while in many domains this may be true, Bloomberg's view was that full automation was far from ideal. This stance conflicts with much (although not all) academic research; the view expressed in many studies is that to tackle difficulty, potential bias and scale in manual fix assessment, automated fix assessment is necessary. Bloomberg introduced a stringent mechanism for accepting or rejecting automatically generated fixes before they were applied. First, a fix should pass all Continuous Integration checks and then it should be verified and reviewed by at least one software engineer as a second line of defence. Research APR tools often discard the fixes that did not pass the verification process [10]. On the other hand, if the verification process at Bloomberg rejects a fix, it will remain recorded and might be offered the next time that the same or a similar bug is encountered.

Fix readability

Ideally, fixes that minimise differences between the fixed program and the buggy program are usually preferred. Fixes that are both simple and readable should be a principle of APR. However, within academic research, there is some debate around this, with some research highlighting the importance of fix readability [11] and other research challenging the idea that understandability is always an appropriate goal [12].

Bloomberg views the readability of a fix and future-proofing of fixes as a fundamental and crucial part of the overall repair process. It is Bloomberg's intention to develop templates for families of repairs and produce an APR system that is configurable; a key principle of that is to develop small and simple fixes. The team has also invested significant time in optimising the user interface "experience". A criticism often levelled by software engineers at software tools is that they lack usability - Bloomberg is also trying to address that aspect of tool usage.

The APR architecture

The underlying technique used at Bloomberg is called "Fixie"; its architecture is illustrated in Figure 1.

Fixie has three different fix types which can be input into the system:

- Off-the-shelf: These fixes are already provided by third-party tools such as clang-tidy[1]; see Figure 2a.
- Custom fixes: Software engineers can also provide fixes which they can apply to other code bases (see Figure 2b). Post-mortems of software outages can identify these fix types. Software engineer-provided fixes can include Abstract Syntax Tree (AST) based fixes using tools such as srcML[2], Clang refactoring tools and simple scripts.
- Fixes automatically learned from version control history: Although still in its infancy, this type of fix has proved to be useful, especially if driven by bugs found from static/dynamic code analysis (see Figure 2c).

"Fixie-**learn**" is the component developed for learning fixes and generating fix patterns automatically from version control history linked to other data sources such as bug/issue tracking systems and CI/CD (Continuous Integration / Continuous Deployment) systems. "Fixie-**store**" is the central

[1] clang.llvm.org/docs/RefactoringEngine.html
[2] www.srcml.org

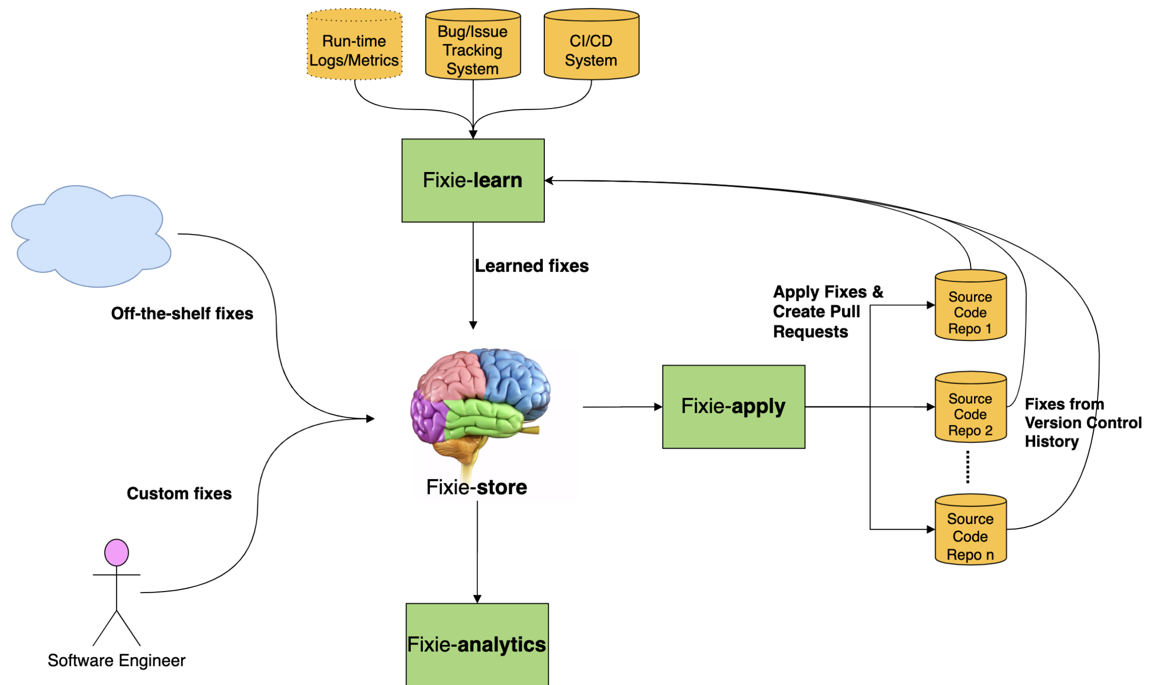

Figure 1: The Fixie architecture

part of the system and stores provided and learned fixes. “Fixie-**apply**” offers fixes to software engineers by creating pull requests (PRs). “Fixie-**analytics**” measures software engineer responses and acceptance of fixes provided by Fixie-apply.

The generation of fix patterns in Fixie-learn is the Bloomberg implementation of the GumTree [13] and anti-unification [14] algorithms (also used by Facebook’s Getafix [1]). Fixie-learn takes a code change (as a commit from version control history) and finds the differences between two ASTs (before and after the code change). To generate a fix pattern, Fixie-learn takes a pair of code changes (two commits), anti-unifies them and extracts the richest AST fix pattern that can reproduce both original code changes. A fix pattern is composed of a before pattern and an after pattern and it is chosen as a fix candidate if the before pattern matches a targeted part of code. The situation requiring attention can be any chunk of code, but is usually in the context of an error code or lint warning, as opposed to a program crash; this can significantly improve the outcome, by filtering the fix patterns to only those learned from such situations. Multiple fix candidates for the same targeted code need to be ranked. Unlike Getafix (where ranking is based on the relevance of the fix patterns to the code changes they were generated from), Fixie-learn ranks the fix candidates according to their success in producing a correct fix in the past. The fix generated from the top fix pattern is then provided as a report to software engineers who view one fix at a time, each of which they can then either accept or reject. In Bloomberg’s opinion, this is practical from a user point of view. Six common bug types were selected for Fixie-learn to learn fixes from and, for each of them, 500 commits that served as examples on how to fix them. Pairing these commits to each other generated around 120k fix patterns. After discarding invalid and duplicated fix patterns and those with a low success rate in the past, the remaining fix patterns produced 5-20 unique fix candidates for the chosen bug types. The deliberate strategy of only attempting to fix well-known, recurring bugs that take up much of software engineers’ time allowed the Fixie tool to be integrated into development processes at Bloomberg in a relatively unobtrusive way.

```
   void function() {
-    int x;
-    char *txt;
-    double d;
+    int x = 0;
+    char *txt = nullptr;
+    double d = NAN;
   }
```

```
-  for (std::set<std::string>::iterator
-                        it = list.begin();
+  for (auto it = list.begin();
               it != list.end(); ++it) {
```

(a) Two off-the-shelf fixes provided by clang-tidy. 1) The tool detects local variables without an initial value and initialises them to zero. 2) The tool suggests modernising the `for` loop to exploit features in a new C++ standard. Software engineers sometimes rejected this fix because the change was incompatible with older compilers still in use.

```
-  if (object.functionReturningEnum()) {
+  if (object.functionReturningEnum() ==
+       namespace::Class::Success) {
   }
```

(b) A custom fix example provided by software engineers. A function whose return type is an enumeration was used as if its return type was Boolean.

```
-  Class::~Class() throw()
+  Class::~Class()
   {
   }
```

```
-  void f(const int x) {
+  void f(const int& x) {
```

```
-  namespace::ClassA objectA(&object1,
                             object2, object3);
+  namespace::ClassA objectA(object2, object3);
```

(c) Three fixes provided by Fixie-learn. 1) Dynamic exception specifications have to be removed from the code when using newer versions of C++. 2) Passing the value of an argument is replaced by passing its reference. 3) A deprecated parameter usage is removed.

Figure 2: Samples of code fixes at Bloomberg. Those in green replaced lines of code marked red.

User feedback

There was initially some critical feedback from software engineers using the tool. Some responded negatively to suggested fixes that, for them, came “out of the blue” and which were perceived as either time-wasting or distracting. Bloomberg’s approach was subsequently shifted to a focus on goal alignment; in other words, integrating Fixie into particular work streams where its benefits would be much clearer. This approach positioned Fixie as helpful rather than intrusive and, most importantly, put the user in control. In a recent migration of build systems in Bloomberg, software engineers around the world got a chance to verify and apply Fixie-suggested code change instead of applying them manually. This led to an acceptance rate of 48% (61/127) and very positive feedback. The general opinion now is that the tool is easy to use and helpful. The tool has also helped software engineers with ideas on how to re-engineer areas of code close to the fix and for promoting a heightened awareness of practices for preventing bugs arising in the first place.

As an interesting side-effect, the language and terminology for describing bugs and their fixes have become more widely understood by software engineers using the tool, aiding software engineer inter-communication. Finally, the tool has freed up time for software engineers to work on other aspects of the code that they would not usually have the time for, such as keeping code *clean*.

APR context and insights

The APR prototype had only minor impact on project management (PM) style or effectiveness at Bloomberg. There was some initial concern about the risk of inserting bugs as part of the APR process. However, software engineers were re-assured by the unobtrusive assistance received from the APR tool, which helped rather than hindered PM. In the COVID-19 climate of working from home, software engineers saw the tool as an important home “assistant” and similar to working closely with another software engineer. APR powered development ran seamlessly with the Agile development strategy at the company. Rather than conflicting with conventional software engineering processes, it was integrated into them without any problems.

Insights into systems at Bloomberg from APR arise from three sources. Firstly, software engineers seemed to develop a greater understanding of how refactoring [15] could be applied as part of the bug fix process, a practice that is often overlooked due to time pressure. Secondly, engineers started to suggest their own fixes to common bugs, effectively extending the reach of the APR tool; this was an unexpected side-effect of the tool. Finally, because APR exposed engineers to different styles of code in the bug fix,

it became evident that engineers were becoming more careful with, and reflective of, their own coding nuances.

Future directions

For future directions, ideally, Bloomberg would like to have several Fixie tools running in parallel. Each would give differing levels of confidence in the solutions offered. For example, one Fixie tool might offer experimental solutions while another tool might offer only well-accepted solutions. The choice as to which to use would depend on the level of risk acceptable at the time. Also, an APR solution may hold the long-term promise of learning about where and why outages arose, a logging system to be established over time and template solutions provided automatically when outages arose. Finally, the APR tool at Bloomberg is currently just a prototype that needs to be developed further and fine-tuned.

Acknowledgments

The study was partly funded by the UK's Engineering and Physical Sciences Research Council (grant number: EP/S005730/1).

REFERENCES

1. J. Bader, A. Scott, M. Pradel, and S. Chandra, "Getafix: Learning to fix bugs automatically," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3360585
2. R. Bavishi, H. Yoshida, and M. Prasad, "Phoenix: automated data-driven synthesis of repairs for static analysis violations," in *European Conference and Symposium on the Foundations of Software Engineering, Tallinn, Estonia, 2019*. ACM, 2019, pp. 613–624.
3. W. Weimer, T. Nguyen, C. L. Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. IEEE, 2009, pp. 364–374.
4. S. Kirbas, B. Caglayan, T. Hall, S. Counsell, D. Bowes, A. Sen, and A. Bener, "The relationship between evolutionary coupling and defects in large industrial software," *J. Softw. Evol. Process.*, vol. 29, no. 4, 2017. [Online]. Available: https://doi.org/10.1002/smr.1842
5. A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott, "Sapfix: automated end-to-end repair at scale," in *International Conference on Software Engineering, Montreal, Canada, 2019*. IEEE / ACM, 2019, pp. 269–278.
6. D. Distefano, M. Fähndrich, F. Logozzo, and P. W. O'Hearn, "Scaling static analyses at facebook," *Commun. ACM*, vol. 62, no. 8, pp. 62–70, 2019. [Online]. Available: https://doi.org/10.1145/3338112
7. C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, "Lessons from building static analysis tools at google," *Commun. ACM*, vol. 61, no. 4, pp. 58–66, 2018. [Online]. Available: https://doi.org/10.1145/3188720
8. M. White, M. Tufano, M. Martinez, M. Monperrus, and D. Poshyvanyk, "Sorting and transforming program repair ingredients via deep learning code similarities," *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb 2019. [Online]. Available: http://dx.doi.org/10.1109/SANER.2019.8668043
9. M. Motwani, S. Sankaranarayanan, R. Just, and Y. Brun, "Do automated program repair techniques repair hard and important bugs?" *Empirical Software Engineering*, pp. 1–47, 11 2017.
10. H. Ye, M. Martinez, and M. Monperrus, "Automated patch assessment for program repair at scale," *CoRR*, vol. abs/1909.13694, 2019. [Online]. Available: http://arxiv.org/abs/1909.13694
11. S. Mechtaev, J. Yi, and A. Roychoudhury, "Directfix: Looking for simple program repairs," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, A. Bertolino, G. Canfora, and S. G. Elbaum, Eds. IEEE Computer Society, 2015, pp. 448–458.
12. M. Monperrus, "A critical review of "automatic patch generation learned from human-written patches": Essay on the problem statement and the evaluation of automatic software repair," in *International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2014, p. 234–242.
13. J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, 2014, pp. 313–324. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642982
14. T. Kutsia, J. Levy, and M. Villaret, "Anti-unification for unranked terms and hedges," *Journal of Automated Reasoning*, vol. 52, no. 2, pp. 155–190, 2014. [Online]. Available: https://doi.org/10.1007/s10817-013-9285-6

15. M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.