# Experience Report: How Effective Is Automated Program Repair for Industrial Software?

Kunihiro Noda, Yusuke Nemoto, Keisuke Hotta, Hideo Tanida, and Shinji Kikuchi
*FUJITSU LABORATORIES LTD., Japan*
{noda.kunihiro, y-nemoto, hotta-keisuke, tanida.hideo, skikuchi}@fujitsu.com

***Abstract*—Recent advances in automated program repair (APR) have widely caught the attention of industrial developers as a way of reducing debugging costs. While hundreds of studies have evaluated the effectiveness of APR on open-source software, industrial case studies on APR have been rarely reported; it is still unclear whether APR can work well for industrial software.**

**This paper reports our experience applying a state-of-the-art APR technique, ELIXIR, to large industrial software consisting of 150+ Java projects and 13 years of development histories. It provides lessons learned and recommendations regarding obstacles to the industrial use of current APR: low recall (7.7%), lack of bug-exposing tests (90%), low success rate (10%), among others. We also report the preliminary results of our ongoing improvement of ELIXIR. With some simple enhancements, the success rate of repair has been greatly improved by up to 40%.**

***Index Terms*—automated program repair, industrial experience report, practical performance**

## I. INTRODUCTION

Automated program repair (APR) has been drawing a great deal of attention in the last decade. A large body of work has led to various bug-fixing patch generation techniques [1]; quite a few real bugs can be fixed using APR tools [2]. While hundreds of existing APR studies have evaluated their techniques with real bugs collected from open-source software (OSS) (e.g., Defects4J [3]), experience reports on industrial applications of APR are scarce [4].

The only APR success story in industry is the case of Facebook, where two APR tools, SapFix [5] and Getafix [6], are integrated into their development workflow. The studies [5], [6] reported that the tools successfully repaired over 40–50% of the null-related bugs/warnings detected by automatically designed test cases or their static code checker. Although this performance is attractive, only null value-related repair is evaluated in the actual development workflow.

To the best of our knowledge, only Naitou et al. reported an industrial case study of more general repair techniques [7] (their targets include various kinds of bugs). They applied two general APR tools [8], [9] to real bugs; however, it resulted in only 1 correct fix (out of 9 bugs). Thus, it is still unclear whether APR tools can work effectively in industry, and more industrial case studies are needed.

This paper presents our industrial experience with applying a state-of-the-art APR tool, ELIXIR [10]. It discusses the actual performance of current APR measured on large industrial software, lessons learned, and the preliminary results of our ongoing improvement of ELIXIR. We analyze large industrial software, consisting of over 150 Java projects (3.5 MLOC), and 6K bugs from 13 years of development histories.

Our industrial case study reveals some challenging problems to address: low repair recall (7.7%), lack of bug-exposing test cases (90% of the bugs), poor success rate (10%), and others. This indicates APR tools might have lower performance and some infeasibility, compared with those in the literature using OSS datasets. It also emphasizes the importance of further improvement of practical aspects of APR techniques.

Also, some enhancements are implemented in ELIXIR; the repair success rate is greatly increased by up to 40%, while our first trial with sampled 20 bugs results in only 2 (10%) correct fixes. We consider there is much room for practical improvement of APR. We hope that our experience contributes to future APR research and industrial applications.

The major contributions of this paper are as follows:

- An industrial experience report on an APR tool targeting various (nonspecific) types of bugs, based on a much larger industrial dataset than the existing report [7];
- Lessons and recommendations from our case study, which reveals poor-performance/infeasibility of current APR;
- Preliminary results of our ongoing enhancements to ELIXIR, which greatly improve repair performance.

## II. BACKGROUND

### A. State-of-The-Art APR Tools and Performance

APR tools first localize bug locations, then generate candidate patches for bug fixing. Finally, they output the patches that pass all test suites (called *plausible*). The types of APR approaches are diverse, ranging from search based [11] and semantics driven [12] to neural machine translation based [13].

Recently, Liu et al. provided a comparison report on the performance of 17 APR tools [14]. Of those, the top four state-of-the-art APR tools with the best performance are shown in Table I. They could repair 21–34 bugs with 50–84% precision among 200+ bugs in Defects4J [3]. As mentioned in Section I, while many studies evaluated APR tools on OSS datasets, industrial ones have been rarely reported.

### B. ELIXIR: Effective Object-Oriented Program Repair

In our industrial case study, we utilize ELIXIR [10], one of the best performing state-of-the-art APR tools listed in Table I.

ELIXIR is a fix patterns-based generate-and-validate (G&V) repair tool. Given a buggy program and test suites, it first

TABLE I
PERFORMANCE OF THE STATE-OF-THE-ART APR TOOLS.

| Project | TBar [14] | SimFix [15] | ELIXIR [10] | CapGen [16] |
|---|---|---|---|---|
| Chart | 9/14 | 4/8 | 4/7 | 4/4 |
| Lang | 5/14 | 9/13 | 8/12 | 5/5 |
| Math | 19/36 | 14/26 | 12/19 | 12/16 |
| Time | 1/3 | 1/1 | 2/3 | 0/0 |
| Total | 34/67 | 28/48 | 26/41 | 21/25 |
| Precision [%] | 50.7 | 58.3 | 63.4 | 84.0 |

Each cell shows #correct/#plausible patches generated. This table is excerpted from the paper by Liu et al. [14]. Only APR tools that could repair over 20 bugs in Defects4J are shown here. Closure and Mockito projects are excluded because ELIXIR and CapGen have not been evaluated on the projects.

localizes the buggy location via spectrum-based fault localization with Ochiai scores, which is a widely used approach in the literature. Then, it generates bug-fixing patches based on several common fix patterns extracted from existing human-written patches (e.g., inserting a null checker, tightening an if condition, etc.). Finally, the first plausible patch, which passes all the existing test cases, is selected as the output.

One key feature of ELIXIR is its repair strategy regarding method invocations (MIs). ELIXIR has rich repair templates (fix patterns) regarding MIs, which are rarely implemented in other APR tools. This greatly contributes to the repair performance for object-oriented programs (OOPs) because bug fixes comprising changes of MIs are very prevalent (over 30–40% of all the one-line bug fixes) in OOPs [10].

A major baneful effect of leveraging rich fix patterns is to cause search space explosion during patch syntheses. Allowing mutation of MIs exponentially increases the number of possible combinations of repair ingredients, making APR infeasible. To deal with that, ELIXIR utilizes a machine-learned model for ranking the candidate patches well. Given the context code of the buggy location and bug reports, the model predicts which patch is more suitable for the location based on several features (e.g., frequencies of identifiers used, token similarity, etc.).

## III. INDUSTRIAL CASE STUDY OF APR

### A. Research Questions and Motivations

**RQ1:** How prevalent are single-statement bug fixes in industrial software development?

First, we investigate the prevalence of single-statement bug fixes. The primary objective of leveraging APR is to reduce debugging costs; thus, we would like to know to what extent APR can (theoretically) contribute to cost savings.

Most state-of-the-art APR tools target single-statement bugs [17]. Although a few tools are capable of fixing multi-hunk bugs [12], [17], the number (classes) of such repairable multi-hunk bugs are very limited. Hence, we investigate the prevalence of single-statement bug fixes in industrial development, which provides an approximate answer to RQ1.

**RQ2:** How many industrial bugs does ELIXIR fix?

The APR success rate is not sufficiently high even for OSS datasets due to the lack of fix patterns, patch overfitting, and other factors. As described in Section I, the actual performance of current APR in industry is still unclear. Thus, we would like to investigate how many of our industrial bugs are fixed by state-of-the-art APR technology.

### B. Subject Software and Data Analyzed

The subject of this case study is large industrial software. It consists of 158 Java projects, 3.5MLOC production code, and 0.5MLOC test code at the latest revision. It utilizes OracleDB for recording business data, JSP & Tomcat for its web interface, and JUnit & DbUnit as testing frameworks.

Apart from the software itself, commit and issue histories are stored in Subversion repositories and Mantis. We analyze over 13 years of large development histories that include 10 individual source repositories, 176K commits, and 20K issues.

### C. Case Study Procedure

**RQ1: Prevalence of single-statement bug fixes**

The Mantis contains multiple kinds and statuses of issues: feature requests, (un)resolved bugs, etc. We first retrieve issues of resolved bugs. Then, we relate each of the issues to bug-fixing commits based on the issue ID in commit messages. Afterward, we manually examine each commit diff to identify whether the change in the commit is a single-statement fix. Note that we consider a multiline change as a single-statement fix if it can be transformed into a semantically-equal single-line change (e.g., Listing 1 can be transformed into Listing 2 by inlining and ignoring a comment). As such, we count the number of single-statement bug fixes in the history.

```
  import java.util.Set;
+ import java.util.TreeSet;
...
-    return db.getBooks();
+    // ensure the book collection is sorted
+    Set<Book> books = db.getBooks();
+    return new TreeSet<>(books);
```

Listing 1. Multiline fix.

```
  import java.util.Set;
...
-    return store.getAllBooks();
+    return new java.util.TreeSet<>(db.getBooks());
```

Listing 2. Single-line fix that is semantically equal to Listing 1.

**RQ2: Practical repair performance of ELIXIR**

We apply ELIXIR to the latest 20 real bugs among all the single-statement bug fixes identified at RQ1, and investigate how many of them are correctly fixed. Note we skip bugs that are difficult to expose by test code (e.g., bugs of GUI layouts). We also evaluate the individual performance of ELIXIR's sub-components, fault localization and patch generation, meaning where each buggy line is ranked and how many of the bugs are fixed under perfect fault localization results.

As for the ELIXIR parameters, up to the top 200 suspicious lines are examined as repair targets for each bug, and up to the top 100 candidate patches are validated for each fix pattern. Assuming the use case of overnight batch processing, we set the timeout of a repair trial for each bug to 12 hours.

### D. Results and Discussion

**RQ1: Prevalence of single-statement bug fixes**

The result of investigating the issue history is as follows:

**6,551** bugs are marked as resolved in Mantis;
**5,857** bugs each are related to at least one bug-fixing commit;

**5,284** bugs each require a fix of at least one production file;
**1,439** bugs each require a fix of only one production *.java;
**406** bugs each require a single-statement fix.

**Lesson 1: Low Recall.** As shown in the above result, only 7.7% (406/5,284) of the bugs involving production-file changes are single-statement fixes. Compared with the fact that Defects4J, which is used in most of the existing studies, contains a large number of single-line fixes (24.8%; 98/395 bugs), the ratio of 7.7% is quite small. ***Single-statement bugs are not prevalent in an actual industrial development history. This means that major state-of-the-art APR tools can repair a very small portion of all the bugs***; the actual recall of major APR tools will be $< 7.7\%$.

The current main streams of APR research are to prevent patch-overfitting and better rank candidate patches; however, ***because the primary objective of integrating APR into software development is to lower debugging costs, future research should emphasize improving recall (as well as precision).*** It is also worth noting that production files contain other file types (*.properties, *.js, etc.) in addition to *.java files; inventing repair techniques for buggy resource files is also an important topic.

### RQ2: Practical repair performance of ELIXIR

We evaluated the repair performance of ELIXIR by applying it to the latest 20 single-statement bugs identified at RQ1.

**Lesson 2: Lack of Bug-Exposing Test Cases.** First, the most critical obstacle to applying ELIXIR is the lack of bug-exposing test cases: only 2 of the 20 bugs are exposed by the existing test methods. A possible reason is that, because writing test code for complex business logic or UI-related scenarios is time-consuming and cumbersome, manual testing tends to be preferred because of tight cost and delivery constraints. This means test-driven repair, the major approach in the literature, is infeasible for 90% of all the bugs.

***The major APR tools assume that there are bug-exposing test cases. This assumption is, however, often invalid in actual industrial software development.*** A recent study [18] also reported that such an assumption is invalid for real bugs from OSS: in a realistic situation, 92% of all the bugs in Defects4J are not exposed by any test cases.

To integrate APR into industrial software development, we might need to provide developers with guidelines that recommend writing a few bug-exposing test cases together with registering a bug report. Also, to resolve or mitigate the issue, it is important to approach automated repair from a different angle; for example, bug report-driven repair [18] or static analysis-based repair [19].

In this case study, we manually add 1–10 test methods (3 methods on average) for each bug that has no bug-exposing test cases. We write new test cases so that the C0 coverage of the method enclosing each bug is close to 100%.

Table II shows the repair performance in this case study. With real (resp. perfect) fault localization results, ELIXIR generates 6 (resp. 5) plausible patches, and only 2 of them correctly fix the bugs (ELIXIR-*R/P* columns in Table II).

TABLE II
RESULTS OF APPLYING ELIXIR TO INDUSTRIAL BUGS.

| | Test Time | | | | ELIXIR | | ELIXIR+ | |
|---|---|---|---|---|---|---|---|---|
| Issue ID | P [s] | EC [s] | WE | Bug Type | R | P | R | P |
| 7498 | 6 | 5 | 175 | W-Meth | N | N | I | **C** |
| 7826 | 3,720 | 119 | 26 | W-Arg | I | N | **C** | **C** |
| 7852 | 16 | 1 | 2 | M-NG | N | N | **C** | **C** |
| 8169 | 277 | 110 | 13 | W-Arg | T | N | **C** | **C** |
| 8183 | 3,720 | 804 | 3 | W-Meth | **C** | **C** | **C** | **C** |
| 8384 | 500 | 196 | 22 | M-Meth | E | N | T | N |
| 9037 | 231 | 92 | 127 | M-Meth | T | N | T | N |
| 12160 | 6 | 1 | 21 | W-Meth | I | N | I | N |
| 18321 | 72 | 2 | 1 | M-Meth | E | N | E | N |
| 18326 | 313 | 2 | 7 | W-Cond | T | N | E | N |
| 18465 | 244 | 52 | 78 | W-Meth | T | N | T | I |
| 18469 | 244 | 28 | 110 | W-Meth | T | I | T | I |
| 19065 | 6 | 2 | 8 | W-Meth | T | N | **C** | **C** |
| 19093 | 4 | 4 | 47 | M-Meth | I | N | I | N |
| 19179 | 37 | 13 | 37 | M-Meth | T | N | T | N |
| 19345 | 5 | 2 | 3 | W-Cond | I | I | **C** | **C** |
| 19598 | 48 | 38 | 3 | W-Arg | N | N | **C** | **C** |
| 19902 | 9 | 9 | 140 | M-Meth | T | N | E | N |
| 20003 | 920 | 61 | 14 | W-Meth | **C** | **C** | **C** | **C** |
| 20038 | error | 1329 | 6 | M-NG | E | I | T | **C** |
| Average | 546 | 144 | 42 | #correct | **2** | **2** | **8** | **10** |
| Median | 72 | 21 | 18 | #incorrect | 4 | 3 | 3 | 2 |
| | | | | #timeout | 8 | 0 | 6 | 0 |
| | | | | #no | 3 | 15 | 0 | 8 |
| | | | | #error | 3 | 0 | 3 | 0 |
| | | | | Correct [%] | **10** | **10** | **40** | **50** |
| | | | | Precision [%] | **33** | **40** | **73** | **83** |

*P* shows the test time of the project enclosing the buggy line while *EC* shows the test time of bug-exposing test classes for each bug.
*WE* shows wasted effort calculated by $H + S/2$. $H$ (resp. $S$) is #lines whose suspicious scores are higher than (resp. the same as) that of the buggy line.
*Bug Type*: *W-* and *M-* mean *wrong* and *missing*, resp.; *Meth*, *Arg*, *NG*, and *Cond* mean *method-call*, *arguments*, *null-guard*, and *condition*, resp.
Four of the columns from the right shows the results of patch generation. *R* (resp. *P*) shows patch generation results with real (resp. perfect) fault localization results. *C*, *I*, *N*, *T*, and *E* mean *correct*, *incorrect*, *no (patches)*, *timeout*, and *error*, resp.
Correct (resp. Precision) is calculated by #correct/#issues (resp. #correct/#plausible).

**Lesson 3: Insufficient Fix Patterns, Ingredients, and Ranking.** Only 2 bugs are correctly fixed by ELIXIR. The rest of the 18 unrepaired bugs are classified into the three types:

(A) *Lack of fix patterns (repair template)*: 2 of the 18 bugs cannot be repaired by ELIXIR because it has no repair templates corresponding to the bugs.

(B) *Insufficient repair ingredients*: 2 of the 18 bugs cannot be repaired because some specific literals (e.g., '2' or '4') are required to fix them. ELIXIR extracts literals only in scope as repair ingredients, and those specific ones are out of scope.

(C) *Insufficient performance of patch ranking strategy*: The rest of the 14 unrepaired bugs are classified into this type. ELIXIR synthesizes candidate patches using ingredients of accessible literals, variables, and methods. Since ELIXIR has rich MI repair templates (described in Section II-B), a large number of candidate patches are generated. Theoretically, ELIXIR can generate correct patches for the 14 bugs; however, those correct ones are difficult to rank in the top 100 due to the overwhelming number of candidates, even though ELIXIR has a neat machine-learned model for ranking them well.

***The competitive repair performance of* ELIXIR *has been demonstrated in the literature for OSS datasets; however, the practical performance on our industrial bug datasets is quite low. This indicates the current* ELIXIR *repair algorithm might strongly overfit the OSS benchmarks.*** A recent study

also reported an issue that repair tools overfit Defects4J [2]. ***Future research should seek and examine many more varied types of real bugs; sophisticating fix patterns, repair ingredients, and ranking strategy based on insights from unseen bugs is required to improve the practicality of APR.***

**Lesson 4: Importance of Test Order and Selection.** G&V repair tools require many times of test executions for fault localization and patch validation. As shown in Table II, while the test time of each bug-exposing class ranges from a few sec. to $\approx$ 20 min., that of each enclosing Java project ranges from 4 sec. to $\approx$ 1 hour; the test time of each enclosing project is much greater than that of each bug-exposing class.

***Compared with OSS datasets, the test time of the industrial software tends to be much longer.*** While it needs 13.5 min. to generate a patch on average for OSS datasets [2], ELIXIR requires 2–3.5 hours to generate a correct patch for our dataset. One of the major reasons for that is the heavy overhead of DB accesses. ***Enterprise applications often involve interactions with outer environments (e.g., DB, network, etc.)***, while existing OSS datasets do not tend to have such properties. Each test that asserts business logic involving such interactions tends to be time-consuming because it needs, for example, connection establishment or data en(de)coding. Preparing stubs or mock objects could mitigate this issue; however, it is not always practically possible due to time or cost limitations (i.e., writing stubs or mock objects for several complicated business objects requires a certain amount of manual effort).

In industrial settings, ***if APR tools simply run entire test cases every time of fault localization and patch validation, it easily exceeds the time budget*** (c.f., it is usually set to 1.5–3 hours in the literature). ***Thus, it needs to carefully select (or design) which test cases should be executed in which order.*** In a debugging phase, developers usually perform impact analysis and select which tests should be executed. APR tools should also perform similar analysis, select a proper subset of test cases, and decide an execution order thereof. For example, first, a candidate patch should be validated only with bug-exposing test classes; then, if those tests do not fail, the patch should be additionally checked with the other test cases.

**Other Practical Issues and Concerns.** Lastly, we list other practical issues and concerns encountered during the case study and discussions with developers, which would be seeds for future APR research.

(A) *Few Opportunities*: G&V repair tools are intended to be integrated into a continuous integration server due to their long execution time. However, developers tend to commit (push) their code after ensuring no test failure; there might be few or no opportunities to trigger the APR tools.

(B) *Slow Response*: Tight schedules are often required in industrial software development. The time budget of several hours for each bug seems to be too long; it might be difficult to integrate an APR tool into a development process involving many daily code changes and commits.

(C) *Difficulty of Review*: Reviewing the patches generated could be time-consuming for developers because no explanations and comments are attached, while human-written patches often include them.

(D) *Multiple Bugs*: In actual software development, multiple bugs often exist simultaneously, which is not handled by current major APR tools.

(E) *Undesirable Side Effects*: Random mutations during repair can cause dangerous unintended behavior: e.g., confidential information might be sent to public servers by mutating variables of server addresses; StackOverflowError caused by mutations could prevent graceful exit, which results in a freeze (an illegal state) of DB management systems.

## IV. PRELIMINARY RESULTS OF IMPROVING ELIXIR

This section describes the preliminary result of our ongoing improvement of ELIXIR. To mitigate the issue in lesson 3, we are implementing an enhanced version of ELIXIR, called ELIXIR+. Current enhancements are threefold: introducing two repair templates and redundancy-based synthesis strategy.

First, we add a repair template to ELIXIR for lesson 3-(A). Both the 2 unrepaired bugs in lesson 3-(A) cause null pointer exceptions (NPEs). Although ELIXIR has a repair template for null-guard insertion, it checks nullness only for variables. The NPEs of the bugs occur because expressions ($\neq$ variables) can be null (e.g., return values of MIs); they cannot be fixed by ELIXIR. Thus, we add a new repair template that inserts null guards for all the expressions in the buggy statement.

The rest of our enhancements are based on the following observations: (1) corrections of the buggy lines often include (parts of) code expressions in other locations in the software (50%; 10/20 bugs); (2) correct code tends to slightly deviate from the buggy code (i.e., edit distance tends to be small). Observation (1) corresponds to the redundancy assumption validated in the literature [20]. It is worth noting that most of those redundant expressions are domain-specific to the subject software; they cannot be obtained from other software.

To mitigate the issues in lessons 3-(B)(C), we introduce a redundancy-based synthesis strategy (RSS). While the original strategy of ELIXIR synthesizes candidate patches with ingredients of accessible literals, variables, and methods, RSS uses as ingredients all the expressions extracted from the method enclosing the buggy statement (including those out of scope).

In the bug-fixing example below, the ingredients of RSS are only the existing expressions in the enclosing method such as *service.isActive()* and *query.contains(...)*, while those of the original strategy are all accessible literals, variables, and methods (including all the accessible fields and methods of *query*, *service*, etc.). Thus, RSS is much more likely to generate the correct patch than the original strategy. It can be considered RSS leverages the knowledge about which method is more likely to be called for the receiver object *service*.

```
- if (query != null) {
+ if (query != null && query.contains("XX-Product")) {
    ...
  }
  ... // tens of lines are here
  if (service.isActive()) {
    ...
    if (query.contains("XX-Product") && hasProfile) {
      ...
```

Listing 3. An example of bug fixing.

In RSS, the patches generated are ranked based on LCS (longest common subsequence) length between original and patched code (from largest to smallest). ELIXIR+ leverages two synthesis strategies: the original one of ELIXIR and RSS. The final list of candidate patches is built by interleaving two lists of patches individually generated from the two synthesis strategies.

Also, we introduce another repair template that simply swaps method arguments (among the same/different expression(s)) in the buggy statement for the issue in lesson 3-(C). This template likely generates candidate patches similar to the original code when mutating method invocations.

The result of the above enhancements is shown in Table II (ELIXIR+ column). Although our enhancements are simple, the performance improvement is remarkable. With the real fault localization results, ELIXIR+ correctly repairs 8/20 (40%) bugs, whereas that of ELIXIR is only 2/20 (10%). With the perfect fault localization results, the success rate rises from 2/20 (10%) to 10/20 (50%).

We consider that the redundancy assumption holds also in industrial software, and thereby redundancy-based patch generation produces a better result. In addition, widening the variety of repair templates will well contribute to the repair performance.

## V. RELATED WORK

APR is a major research topic in the software engineering area; hundreds of papers on APR have been published [1]. While many studies reported on applying APR tools to OSS, industrial experience reports on APR are very few [4].

Two APR tools, SapFix [5] and Getafix [6], are integrated into the Facebook development workflow. SapFix is an end-to-end repair tool: first, it detects latent NPEs with tests automatically designed by Sapienz; then, it tries to repair them via mutation or fix templates, resulting in $\approx 50\%$ correct fixes [5]. Getafix is a repair tool that learns fix patterns from bug-fixing histories. Unlike G&V repair, it utilizes the static analyzer, Infer, for latent bug detection and patch validation. Over 40–60% of null-related bugs are correctly fixed in Facebook [6].

Naitou et al. [7] reported an industrial application of two general APR tools, ASTOR [8] and NOPOL [9]. Of 327 industrial bugs to investigate, they applied the APR tools to 9 bugs, resulting in only 1 correct fix. They also reported some barriers to the industrial use of APR; for instance, only a small portion of the bugs can be repaired by program-code mutation (i.e., other types of files need changing). It indicates the difficulty of applying general APR tools to industrial software and the immatureness of current APR techniques.

Apart from industrial reports, current main streams of APR research are to prevent patch overfitting and better rank candidate patches. A major approach to overfitting prevention is to leverage test case generation [21]. As for better ranking, the types of approaches are diverse: e.g., machine learning-based [10], similarity-based [15], [16], etc.

A recent study reported that the issue of lacking bug-exposing test cases exists also in OSS [14]. To deal with that, new APR approaches from different angles are required, such as bug report-driven [14], static analysis-based [19], etc.

## VI. CONCLUSION

This paper reported our experience applying ELIXIR, a state-of-the-art APR tool, to large industrial software. Our case study revealed several critical obstacles to the industrial use of APR: low recall, lack of bug-exposing tests, and poor success rate, among others. Current APR techniques still have several immature aspects for practical industrial deployment; it needs further improvement of the practicality of APR techniques.

We also presented the preliminary results of our ongoing improvement efforts. ELIXIR+, an enhanced version of ELIXIR, additionally leverages new repair templates and a redundancy-based synthesis strategy based on the insights from our first trial. The enhancements are simple but contribute substantially to repair performance, increasing the success rate of repair from 10% up to 40%.

We hope this report contributes to future research in the APR community.

## REFERENCES


[1] L. Gazzola *et al.*, "Automatic software repair: A survey," *IEEE Trans. Softw. Eng.*, vol. 45, no. 1, pp. 34–67, 2019.

[2] T. Durieux *et al.*, "Empirical review of Java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts," in *FSE*, 2019, pp. 302–313.

[3] R. Just *et al.*, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *ISSTA*, 2014, pp. 437–440.

[4] M. Monperrus, "The living review on automated program repair," HAL/archives-ouvertes.fr, Tech. Rep. hal-01956501, 2018.

[5] A. Marginean *et al.*, "Sapfix: Automated end-to-end repair at scale," in *ICSE-SEIP*, 2019, pp. 269–278.

[6] J. Bader *et al.*, "Getafix: Learning to fix bugs automatically," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, pp. 159:1–159:27, Oct. 2019.

[7] K. Naitou *et al.*, "Toward introducing automated program repair techniques to industrial software development," in *ICPC*, 2018, pp. 332–335.

[8] M. Martinez and M. Monperrus, "ASTOR: A program repair library for Java (demo)," in *ISSTA*, 2016, pp. 441–444.

[9] J. Xuan *et al.*, "Nopol: Automatic repair of conditional statement bugs in Java programs," *IEEE Trans. Softw. Eng.*, vol. 43, no. 1, pp. 34–55, Jan. 2017.

[10] R. K. Saha *et al.*, "Elixir: Effective object-oriented program repair," in *ASE*, 2017, pp. 648–659.

[11] C. Le Goues *et al.*, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *ICSE*, 2012, pp. 3–13.

[12] S. Mechtaev *et al.*, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *ICSE*, 2016, pp. 691–701.

[13] Z. Chen *et al.*, "SEQUENCER: Sequence-to-sequence learning for end-to-end program repair," *IEEE Trans. Softw. Eng.*, 2019.

[14] K. Liu *et al.*, "Tbar: Revisiting template-based automated program repair," in *ISSTA*, 2019, pp. 31–42.

[15] J. Jiang *et al.*, "Shaping program repair space with existing patches and similar code," in *ISSTA*, 2018, pp. 298–309.

[16] M. Wen *et al.*, "Context-aware patch generation for better automated program repair," in *ICSE*, 2018, pp. 1–11.

[17] S. Saha *et al.*, "Harnessing evolution for multi-hunk program repair," in *ICSE*, 2019, pp. 13–24.

[18] A. Koyuncu *et al.*, "iFixR: Bug report driven program repair," in *FSE*, 2019, pp. 314–325.

[19] R. Bavishi *et al.*, "Phoenix: Automated data-driven synthesis of repairs for static analysis violations," in *FSE*, 2019, pp. 613–624.

[20] E. T. Barr *et al.*, "The plastic surgery hypothesis," in *FSE*, 2014, pp. 306–317.

[21] Z. Yu *et al.*, "Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the Nopol repair system," *Empirical Software Engineering*, vol. 24, no. 1, pp. 33–67, Feb 2019.