Elixir: An Automated Repair Tool for Java Programs

Ripon K. Saha, Hiroaki Yoshida
Mukul R. Prasad
Fujitsu Laboratories of America, Inc., USA
{rsaha,hyoshida,mukul}@us.fujitsu.com

Susumu Tokumoto, Kuniharu Takayama
Isao Nanba
Fujitsu Laboratories Ltd., Japan
{tokumoto.susumu,kuniharu,namba}@jp.fujitsu.com

ABSTRACT

Object-oriented (OO) languages, by design, make heavy use of method invocations (MI). Unsurprisingly, a large fraction of OO-program bug patches also involve method invocations. However, current program repair techniques incorporate MIs in very limited ways, ostensibly to avoid searching the huge repair space that method invocations afford. To address this challenge, in previous work, we proposed a generate-and-validate repair technique which can effectively synthesize patches from a repair space rich in method invocation expressions, by using a machine-learned model to rank the space of concrete repairs. In this paper we describe the tool Elixir that instantiates this technique for the repair of Java programs. We describe the architecture, user-interface, and salient features of Elixir, and specific use-cases it can be applied in. We also report on our efforts towards practical deployment of Elixir within our organization, including the initial results of a trial of Elixir on a project of interest to potential customers. A video demonstrating Elixir is available at: https://elixir-tool.github.io/demo-video.html

CCS CONCEPTS

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

KEYWORDS

Automatic Program Repair, Machine Learning, OOP

ACM Reference Format:
Ripon K. Saha, Hiroaki Yoshida Mukul R. Prasad and Susumu Tokumoto, Kuniharu Takayama Isao Nanba. 2018. Elixir: An Automated Repair Tool for Java Programs. In *ICSE '18 Companion: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3183440.3183487

1 INTRODUCTION

Research on automatic program repair (APR) is motivated by the strong need to automate the current practice of manually debugging and patching software bugs, which places substantial costs on the software development process. This work targets object-oriented (OO) languages, which dominate today's programming landscape, constituting 4 of the top 5 languages on the TIOBE Index [7].

The principle of *encapsulation*, which is central to object-oriented language design [1], makes the method invocation (MI) construct the de-facto unit of data access and computation in OO programs. In fact, according to an empirical study on three popular Java projects we reported in [6], as many as 77% of *one-line* bug-fixes made during the lifetime of each of these projects involved a change to or insertion of a method invocation. This statistic suggests that meaningful repair of Java program bugs should incorporate repair and synthesis of method invocations, in a comprehensive manner. However, previous APR techniques have employed MI-related mutations in a very limited way, if at all, in synthesizing repairs [3, 9]. A plausible reason for this is the combinatorial explosion in the repair space that could result from examining expressions formed by all possible objects and method invocations in scope.

In [6] we presented a generate-and-validate (G&V) repair technique for object-oriented programs and its software prototype Elixir, instantiated for Java programs. A key innovation in Elixir is the aggressive use of method invocations (or method calls), on par with local variables, fields, and constants, to construct more expressive *repair expressions*, that go into synthesizing patches. The ensuing enlargement of the repair space is effectively tackled by using a machine-learned model to rank concrete repairs. The machine-learned model relies on four features derived from the *repair context*, *i.e.,* the code surrounding the potential repair location, and from the bug report (if one is present). Intuitively, these features serve to characterize the program elements (*e.g.,* variables, objects, field, methods) that go into patches, and by extension the concrete patch candidates themselves, forming the basis for ranking patch candidates. A key contribution of Elixir is the choice and specific incarnation of the features used, in the context of a machine-learned model used to guide a program repair technique. Also novel is the insight that such a technique can effectively navigate a large and rich repair space to fix bugs involving method invocations in object-oriented programs, specifically Java.

This paper addresses the implementation and deployment of the Elixir tool. Specifically, it makes the following contributions:

- It describes the architecture and user-interface of Elixir (Sec. 4).
- It discusses two specific use-cases of Elixir (Sec. 3).
- It reports on initial results from our efforts towards practical deployment of Elixir within our organization (Sec. 6).

2 TECHNIQUE

Elixir implements a generate-and-validate style (or search-based) repair technique [3–5, 8], for Java programs. Given a buggy program, a test suite (having at least one bug reproducing test case), and optionally a bug report, Elixir produces a patch that passes all the test cases, *i.e.,* fixes the bug. As shown in Figure 1, Elixir works in four steps namely, (A) bug localization, (B) candidate patch generation, (C) ranking and selection, and (D) patch validation.

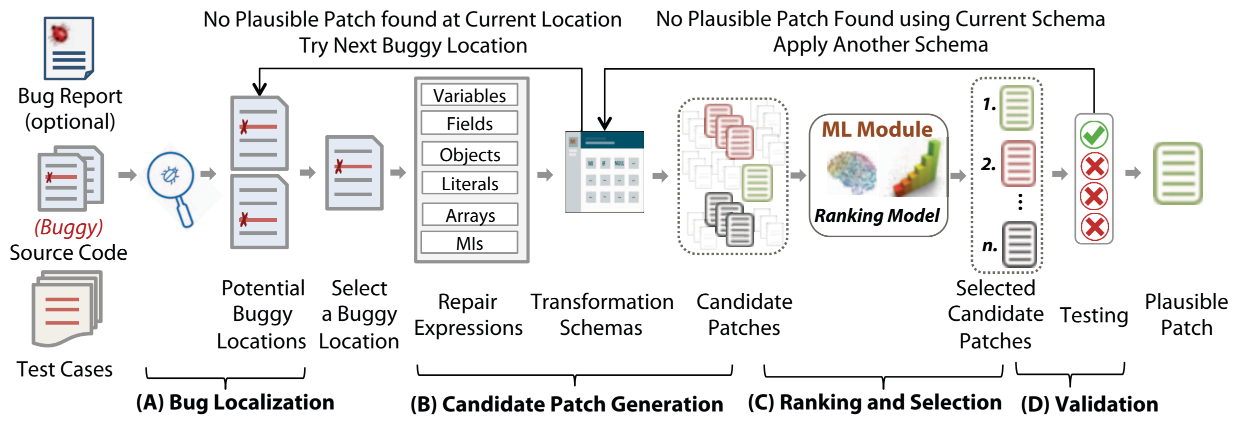


Figure 1: Overview of Elixir.

(A) Bug localization. This steps returns a ranked list of suspicious statements from the buggy program, which constitute potential repair locations. These repair locations are processed, in order, by the subsequent steps, effecting various repairs, until a patch passing all test cases is found or the tool times out.

(B) Candidate patch generation. For each potential repair location this step generates a population of candidate patches (repairs) that could potentially be applied at that location to fix the bug. Elixir generates these candidates by instantiating each of a set of 8 program transformation schemas with expressions, called *repair expressions*. These expressions are generated by a specific context-free grammar using program elements, such as local variables, constants, objects, method and fields of these objects, *etc.,* in scope at the repair location. The program transformation schemas and the grammar for repair expressions are described in [6].

Many of the schemas used in Elixir mirror earlier works on program repair. However, a key distinguishing feature of Elixir is the rich grammar of repair expressions it uses, with objects and MIs on objects instantiated on par with local variables or constants. This allows for richer mutations of existing method invocations as well as Boolean predicates (*e.g.,* in conditional or return statements). Elixir also includes a schema to synthesize and insert a new MI, as part of an existing expression or as a complete new statement. This significantly expands the repair space as well.

(C) Ranking & selection. This step ranks the population of candidate patches produced earlier, using a machine-learned model. Only the top n ranked candidates are then validated against the test suite, rather than the entire (potentially very large) population or an arbitrarily selected subset. This ranking is a key innovation in Elixir and allows it to effectively navigate a rich repair space incorporating MI-based repair expressions.

A *logistic regression* model, trained offline in advance, assigns to each candidate patch a probability score that reflects its likelihood of being *the* correct patch. This likelihood is computed based on the program elements constituting the patch, using four simple but potent features derived from the *repair context, i.e.,* the code surrounding the potential repair location, and from the bug report (if one is present). The features characterize a given program element, quantifying (1) how frequently the program element has been used in the current context, (2) the distance of the place of last use from the repair location, (3) whether *related* program elements have been used in the repair context, and (4) if the program element, or related ones, have been referenced in the bug report.

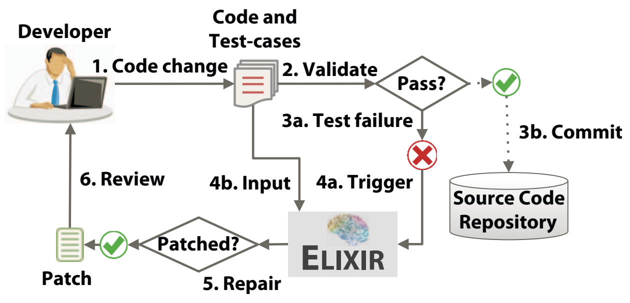


Figure 2: Repair of Development Bugs.

(D) Patch validation. This step validates each selected candidate patch against the complete suite of test-cases. A patch passing all tests is returned as a *plausible patch, i.e.,* a potentially viable repair. Otherwise the search moves on to the next candidate.

3 USAGE SCENARIOS

Elixir can be used in various ways to help developers fix bugs. In this paper, we discuss two concrete scenarios.

3.1 Repair of Development Bugs

Detecting bugs during software development (either in the implementation of a new feature or in a refactoring) is a frequent scenario since it is hard to write bug-free code in the first attempt. For a given development activity, once the developer has a valid failing test, she can use Elixir to get automated repair assistance through an IDE or command-line. As Figure 2 shows, Elixir takes the buggy source code and failing test(s) and tries to generate a patch that passes all the tests. If Elixir generates a patch, the developer can review it, and apply the patch to the codebase if it is correct.

3.2 Repair of Field Bugs

Once a bug is reported from the field during the operation of a software, first the bug triager verifies if the bug is a valid bug and if the provided information is correct. In case of any inconsistencies,

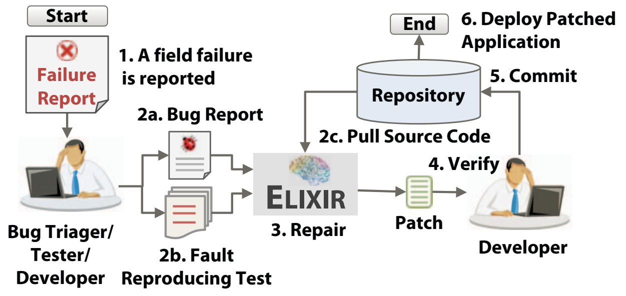

Start
Failure Report
1. A field failure is reported
2a. Bug Report
Bug Triager/ Tester/ Developer
2b. Fault Reproducing Test
ELIXIR
3. Repair
2c. Pull Source Code
Repository
End
6. Deploy Patched Application
5. Commit
4. Verify
Patch
Developer

Figure 3: Repair of Field Bugs.

the bug triager edits the bug report. Sometimes the bug triager or a tester may also try to reproduce the bug to make these initial assessments. Then the bug is assigned to an appropriate developer with or without the failing tests. In case, there is no failing test, the assigned developer writes the failing tests. Then the developer can use Elixir to fix the bug automatically (Figure 3). Elixir is generally more effective if a meaningful bug report is provided.

4 TOOL DESCRIPTION

Implementation. Elixir has been developed from the scratch, as an Eclipse-Maven project, with about 17K lines of Java code and 1K lines of Bash script code. As shown in Figure 4, the Elixir Core Engine is comprised of five major components: i) a fully automatic bug localization system to identify suspicious program locations, ii) an AST transformer to modify programs, iii) a machine learning system to select potential patches, iv) an in-memory compilation system to compile an edited program without storing it, and v) a test analyzer to run test cases automatically and to analyze the test results. Elixir makes use of several open-source libraries for performance and quality. More specifically, Elixir uses the ASM byte code library (http://asm.ow2.org/) to collect code coverage. It uses the Spoon library (http://spoon.gforge.inria.fr/) to build the source code, and to construct and modify the AST of a Java class. The Weka toolkit (https://www.cs.waikato.ac.nz/ml/weka/index.html) is used for machine learning. After applying a repair schema, Elixir leverages `javax.tools` for in-memory compilation. Finally, Elixir uses JUnit APIs to run and analyze the test cases programmatically. Currently, Elixir can be used in two ways: i) as an eclipse plug-in, and ii) as a command-line tool.

Eclipse Plug-in. As an Eclipse plug-in, Elixir can be used in an interactive fashion. Therefore, it is suitable for a development setting. As Figure 5 shows, the Elixir plugin provides a dedicated menu for the developer, which is easy to use, and easily extensible, to provide more features in the future. Currently, a developer can execute Elixir by clicking the "Generate Patch" menu item. During execution, Elixir provides informative log messages in the *Console View* so that developers know what is going on underneath. This is a background process. So developers can continue other work if they want. Once Elixir finds a patch, it presents the patch in a *Patch View* so that the developer can easily see what changes Elixir suggested. Finally the developer can review and apply the suggested patch to the codebase, with just a single button-click.

Command Line Tool. Command-line mode is suitable when Elixir is automatically triggered locally or remotely when a test

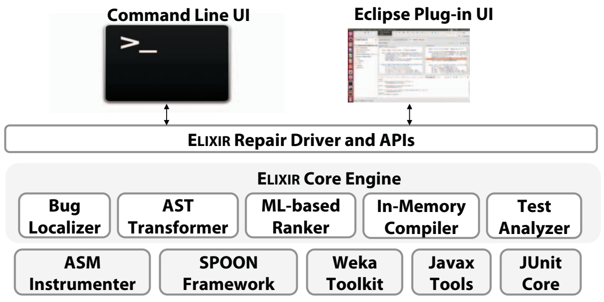
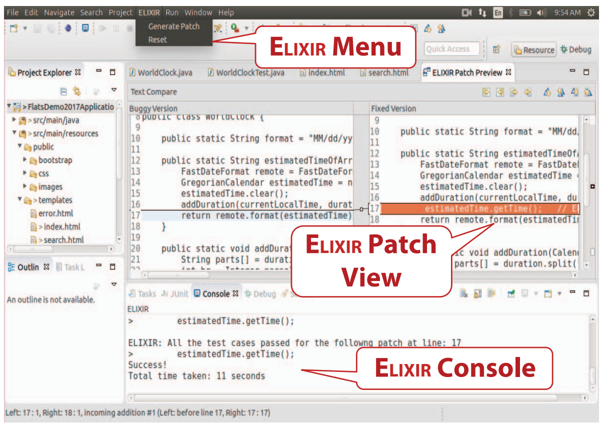


Figure 4: Elixir Software Architecture.



```
public class WorldClock {

    public static String format = "MM/dd/yy

    public static String estimatedTimeOfArr
        FastDateFormat remote = FastDateFor
        GregorianCalendar estimatedTime = n
        estimatedTime.clear();
        addDuration(currentLocalTime, durat
        return remote.format(estimatedTime)
    }

    public static void addDura
        String parts[] = durat
```

```

    public static String format = "MM/dd,

    public static String estimatedTimeOf
        FastDateFormat remote = FastDate
        GregorianCalendar estimatedTime
        estimatedTime.clear();
        addDuration(currentLocalTime, du
         estimatedTime.getTime();   // E
        return remote.format(estimatedTi
```

```
ELIXIR
>       estimatedTime.getTime();

ELIXIR: All the test cases passed for the followng patch at line: 17
>       estimatedTime.getTime();
Success!
Total time taken: 11 seconds
```

Left: 17 : 1, Right: 18 : 1, incoming addition #1 (Left: before line 17, Right: 17 : 17)

Figure 5: Elixir Eclipse Plugin.

case fails either in continuous integration or regression testing. Elixir can be configured easily through a property file and run fully automatically through a Bash script.

5 EVALUATION

The evaluation of Elixir was conducted on two large bug datasets, namely the popular dataset Defects4J [2] and the new large-scale dataset Bugs.jar (https://github.com/bugs-dot-jar/bugs-dot-jar) introduced in [6]. The Defects4J evaluation spans the 4 subjects commonly used in other evaluations (Table 1) and specifically all 82 bugs with single hunk developer patches in these subjects. The Bugs.jar evaluation spanned 7 subjects (Table 2) and 127 one-hunk bugs. All experiments were run on Intel(R) Core(TM) i7-4790 3.60GHz CPUs, with 4GB memory, under Ubuntu 14.04 LTS and Java 7.

We evaluate each tool in terms of number of correct and incorrect patches produced, where each produced patch should have passed all the given tests. A *correct* patch is semantically equivalent to the developer-provided patch, based on a manual examination and consensus among all authors. Otherwise the patch is *incorrect*.

Elixir-Baseline. This is a simplified version of Elixir, built on the same platform. It uses the same repair schema as Elixir, instantiated in the same order. However, it uses simpler repair expressions, following existing tools, such as PAR [3], ACS [8], and HD-Repair, rather than Elixir's MI-rich expressions. Also, it iterates over patch candidates heuristically, rather than guided by a machine-learned model, as in Elixir.

Table 1: Comparison with existing techniques

Subject	C.Math	C.Lang	Joda-Time	JFreeChart	Total
Elixir	12/7	8/4	2/1	4/3	**26/15**
ACS	12/4	3/1	1/0	2/0	18/5
HD-Repair	6/NR	7/NR	1/NR	2/NR	16(10*)/NR
NOPOL	1/20	3/4	0/1	1/5	5/30
PAR′	2/NR	1/NR	0/NR	0/NR	3/NR
jGenProg	5/13	0/0	0/7	0/2	5/22

Each cell presents (Correct/Incorrect) patches. NR=Not Reported.
* HD-Repair generated correct patches for 16 defects, but only 10 were ranked first [8]. All other tools terminate at the first plausible patch.

Table 2: Patch Generation (Correct/Incorrect) on Bugs.jar

Project	In Sample	Elixir	Elixir-Baseline
Accumulo	10	1/0	1/0
Camel	16	2/1	1/0
Commons Math	21	8/3	6/4
Flink	7	2/0	1/0
Jackrabbit Oak	31	3/6	2/4
Maven	5	0/0	0/0
Wicket	37	6/7	3/8
Total	127	22/17	14/16

Elixir vs. other tools. Table 1 compares the performance of Elixir against 5 state-of-the-art G&V repair tools, namely jGenProg, NOPOL, PAR', HD-Repair, and ACS, on the 82 one-hunk Defects4J bugs. The results for the competitor tools are drawn from the respective papers. Overall, Elixir produced 26 *correct* patches, which is the highest on Defects4J. ACS comes in second with 18, while all other tools generate 10 or less. Note that although HD-Repair generated 16 patches it ranked the correct patch first for only 10 of them. Thus, Elixir performs significantly better that other leading tools, in terms of generating correct patches, on Defects4J.

Elixir on Bugs.jar. Table 2 presents the results of Elixir vs. Elixir-Baseline on the 127 bugs from Bugs.jar. We observe that Elixir generates 22 correct patches, vs. only 14 by Elixir-Baseline, *i.e.,* 57% more. This demonstrates that Elixir's superior performance is replicated on Bugs.jar as well. A more comprehensive evaluation on various features of Elixir can be found in [6].

Limitation. Like other G&V techniques, Elixir requires a fault reproducing test to generate a patch, and the quality of the generated patch highly depends on the quality of the test suite. Also, Elixir currently generates only one-hunk patches.

6 TOOL DEPLOYMENT

In an effort to promote the practical adoption of Elixir within our organization we engaged with one specific product division that was facing challenges in utilizing external open source software (OSS) components in their product development. Specifically, the team shared that bugs in the OSS components that were frequent and laborious to understand or fix, particularly in the face of regular updates to the OSS. Fixes from the OSS developers often took weeks, even months. They were interested in seeing if Elixir could automatically fix bugs in such OSS components. They presented Apache ZooKeeper (https://zookeeper.apache.org/) as an OSS project of interest, as a challenge to Elixir.

ZooKeeper is a widely used centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. To apply Elixir, we extracted all reproducible, currently fixed bugs in the ZooKeeper's project history. This yielded a set of 84 bugs, including 16 one-hunk bugs. After minimal changes to Elixir to account for the build system of ZooKeeper, Elixir could be run on the 16 bugs, producing patches for 12 of them, including 10 correct patches. By comparison, Elixir-Baseline could only produce 5 patches, with only 3 correct ones. Encouraged by this result the customer team is continuing discussions with us for a more extensive online trial of Elixir.

7 RELATED WORK

Search-based repair. This is the most common class of repair techniques, pioneered by GenProg [4], which used genetic programming to search a space of repair mutations. Subsequent techniques made contributions to the *search algorithm*, definition of the *repair space* [3, 8], and to the patch generation efficacy, through *search pruning techniques* [5]. Elixir's main contribution is the substantially expanded use of MIs in repairs, which previous techniques use in very limited ways. Elixir uses a machine-learned model to effectively search the expanded repair space.

AI in program repair. Like Elixir, Prophet [5] also uses a machine-learned (ML) model derived from previous human patches to prioritize candidate repairs. However, Prophet attempts to guess the complete correct patch directly while Elixir uses ML to identify the program elements constituting the correct patch, and with better repair outcomes (85% increase in repairs vs. Prophet's 25%).

A more complete discussion of related work can be found in [6].

8 CONCLUSIONS

In [6] we proposed a generate-and-validate repair technique for object-oriented programs, that can effectively navigate a huge repair space, rich in method invocation expressions, and synthesize patches, by using a machine-learned model to rank the concrete repairs. In this paper we described a tool, Elixir that instantiates this technique for Java programs, including its architecture, user-interface, and two specific use-cases of the tool. We also reported some initial results from our efforts towards practical deployment of Elixir within our organization.

REFERENCES

[1] Grady Booch. 2004. *Object-Oriented Analysis and Design with Applications (3rd Edition).* Addison Wesley Longman Publishing Co., Inc.
[2] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *ISSTA*.
[3] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic Patch Generation Learned from Human-written Patches. In *ICSE*.
[4] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each. In *ICSE*.
[5] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *POPL*.
[6] Ripon K. Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R. Prasad. 2017. Elixir: Effective Object-Oriented Program Repair. In *ASE*.
[7] TIOBE Index. 2017. http://www.tiobe.com/tiobe-index/. (October 2017).
[8] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise Condition Synthesis for Program Repair. In *ICSE*.
[9] 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 TSE* 43, 1 (Jan 2017).