ELIXIR: Effective Object-Oriented Program Repair

Ripon K. Saha, Yingjun Lyu*, Hiroaki Yoshida, Mukul R. Prasad
Fujitsu Laboratories of America, Sunnyvale, CA, USA
*University of Southern California, Los Angeles, CA, USA
rsaha@us.fujitsu.com, *yingjunl@usc.edu, hyoshida@us.fujitsu.com, mukul@us.fujitsu.com

***Abstract*—This work is motivated by the pervasive use of method invocations in object-oriented (OO) programs, and indeed their prevalence in patches of OO-program bugs. We propose a generate-and-validate repair technique, called ELIXIR designed to be able to generate such patches. ELIXIR aggressively uses method calls, on par with local variables, fields, or constants, to construct more expressive repair-expressions, that go into synthesizing patches. The ensuing enlargement of the repair space, on account of the wider use of method calls, is effectively tackled by using a machine-learnt model to rank concrete repairs. The machine-learnt model relies on four features derived from the program context, *i.e.,* the code surrounding the potential repair location, and the bug report. We implement ELIXIR and evaluate it on two datasets, the popular Defects4J dataset and a new dataset Bugs.jar created by us, and against 2 baseline versions of our technique, and 5 other techniques representing the state of the art in program repair. Our evaluation shows that ELIXIR is able to increase the number of correctly repaired bugs in Defects4J by 85% (from 14 to 26) and by 57% in Bugs.jar (from 14 to 22), while also significantly out-performing other state-of-the-art repair techniques including ACS, HD-Repair, NOPOL, PAR, and jGenProg.**

I. INTRODUCTION

As software applications continue to grow in size and complexity, and fuel the development of new application domains, such as cloud computing, big-data analytics, mobile computing, and software-defined networks, they inevitably produce a corresponding increase in the number of software bugs, and ultimately in the cost of fixing these bugs. For example, a study from the University of Cambridge showed that, as of 2013, the global cost of debugging software had risen to $312 billion annually [1]. Further, the research found that, on average, software developers spend 50% of their programming time finding and fixing bugs. Automatic software repair techniques have the potential to mitigate some of these costs and thereby increase developer productivity.

Object-oriented (OO) languages dominate the programming landscape today. In fact, 4 of the top 5 languages on the TIOBE Index [2], namely Java, C++, C#, and Python, are object-oriented or bear at least some object-oriented features. However, most techniques for automatic program repair [3], [4], [5], [6], [7], [8], [9], [10], [11], with a few notable exceptions [12], [13], [14], [15], have been developed in the context of C programs, *i.e.,* procedural programs. Studies have shown [16] that bug-patterns can be language specific. Thus, there is a strong need to develop automatic repair techniques targeting object-oriented programs.

One of the core principles of object-oriented language design is the notion of *encapsulation* [17], whereby the internal data representation of a class (object) and its implementation of operations is hidden from external users of the class. External objects can only access this data and operations through the public methods of the class (object). Thus, the construct of a method invocation (MI) (used inter-changeably with the term *method call* in this paper) on an object, constitutes the basic unit of data access and computation in object-oriented programs. In fact, according to an empirical study we conducted on three popular Java projects (discussed in Section II-A), as many as 57% of program statements in each of these applications contain one or more method invocations. Further, according to the same study, 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. These data points demonstrate the need to incorporate repair and synthesis of method invocations, in a comprehensive manner, in the repair of bugs in object-oriented programs. For concreteness, the rest of the paper uses Java as a representative of OO-languages. However, the discussion would be equally applicable to other OO-languages, such as C++.

A number of program repair techniques, proposed for Java programs, like PAR [12], NOPOL [13], HD-Repair [14], and ACS [15], in fact manipulate method invocations in their repairs. However, this is done through very specific schemas and with tight restrictions. For instance, NOPOL is the only tool that synthesizes (*i.e.,* creates from scratch) method calls, but only on manually specified, side-effect-free, parameter-free method calls and only as guards of inserted if-conditions. PAR replaces names or parameters of method calls but only with other names or expressions appearing in other method calls *in the same method*. A plausible reason for such restrictions, documented in a recent work by Martinez and Monperrus [18], is the combinatorial explosion that would result from expanding the repair space to include a much wider scope of method call modifications and insertions. Figure 2 shows a simple example of this, using a bug fix from the *Apache Commons Lang* project, that is correctly patched by our proposed tool, ELIXIR. As shown, the patch consists of a single method invocation. However, for constructing such a method invocation, that is correctly typed and in scope, there are more than 800 concrete candidates! Obviously, a repair approach cannot afford to iterate through each of them.

The second author was an intern at Fujitsu Labs of America, CA, USA during the construction of Bugs.jar.

978-1-5386-2684-9/17/$31.00 © 2017 IEEE

ASE 2017, Urbana-Champaign, IL, USA
Technical Research

This work proposes a generate-and-validate repair technique, called ELIXIR, developed for the repair of object-oriented programs. A key innovation in ELIXIR is the aggressive use of 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. The features describe a given identifier, which could represent a local variable or object, a constant, or a method call. In particular the features quantify (1) how frequently the identifier has been used in the current context, (2) the distance of the place of last use from the repair location, (3) whether "similarly named" tokens have been used in the repair context, and (4) if the identifier or sub-tokens thereof have been referenced in the bug report (if one is present). Variants of some of these features have been used in heuristics employed in code recommendation [19], [20], bug localization [21], [22], and program repair [23]. However, a key contribution of our work is the choice and specific incarnation of those features in the present context, and their use in a machine-learned model used to guide a program repair technique. Also novel is the insight that such a technique can effectively navigate a huge repair space to fix bugs involving method invocations in OO-programs, specifically Java.

We implement and evaluate ELIXIR on two datasets, the popular Defects4J dataset and a new dataset Bugs.jar created by us, and against two baseline versions of our technique, as well as five other tools/techniques representing the state of the art in program repair. Our evaluation shows that ELIXIR is able to increase the number of correctly repaired bugs in Defects4J by 85% (from 14 to 26) and by 57% for Bugs.jar (from 14 to 22), while also significantly out-performing other state-of-the-art repair techniques including ACS [15], HD-Repair [14], NOPOL [13], PAR [12], and jGenProg [41]. This paper makes the following key contributions:

- **Empirical study:** An empirical study highlighting the prevalence of method invocations in patches of Java programs, as a motivation for our proposed technique.
- **Technique:** A novel technique, ELIXIR that aggressively employs method invocations in constructing repairs for Java programs, and object-oriented programs in general.
- **Implementation:** An implementation of ELIXIR in our in-house Java repair framework, along with two baseline versions of ELIXIR.
- **Dataset:** A new, large dataset of 1,158 bugs and patches, Bugs.jar, made available to the research community at [24], to complement existing datasets like Defects4J.
- **Evaluation:** A comprehensive evaluation of ELIXIR on two datasets, Defects4J and Bugs.jar, and against seven competing techniques, including two baseline versions of ELIXIR and five external tools/techniques, ACS, HD-Repair, NOPOL, PAR, and jGenProg.

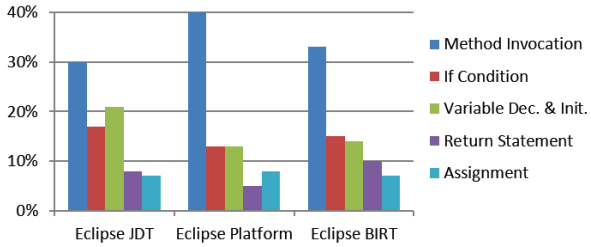

Fig. 1. Distribution of Bug-Fixing Changes. Based on 1186, 1031, and 985 one-line bug fixes in Eclipse JDT, Platform, and BIRT projects respectively.

II. MOTIVATION

This section demonstrates the prevalence of method invocations (MI) and MI-related bugs, in Java programs, through an empirical study and two real-world motivating examples.

A. An Empirical Study on the Method Invocation Construct and Its Relevance to Bugs in Java Programs

The construct of method invocation (MI) is fundamental to orchestrating data access and computation in OO-programs, and indeed for enforcing key OO features, such as encapsulation. However, it is natural to ask if real-world OO-programs (*e.g.,* Java applications) demonstrate a *quantitatively greater use* of MIs than procedure-oriented programs (*e.g.,* C). To investigate this empirically, we selected three Java projects: Eclipse JDT, Platform, and BIRT, which are popular Java projects used in bug localization research [22] and the ManyBugs [25] benchmarks, widely used in C program repair research, representing C applications. The Java projects were intentionally chosen to be distinct from our experimental datasets (Defects4J and Bugs.jar), to guard against learning bias. We parsed all the source code files of both Java and C applications, excluding test cases, and counted the fraction of executable statements having an MI or a function call. Our results show that, on average, 57% of statements in each of the Java applications have an MI, compared to only 33% for the C programs. Thus, this study, while decidedly limited in scope, supports the hypothesis that Java programs use MIs substantially more than C programs.

We further analyzed all the one-line bug-fixes in each of the three Java projects to investigate how often MIs appear in those patches. We focus on one-line bug-fixes since current automatic program repair tools mainly target such fixes. We used ChangeDistiller [26] to extract the one-line bug-fixes throughout each project's history, and automatically classified each patch into one of a few *mutually exclusive* categories based on the type of that statement, such as method invocation, if condition, variable declarations and initializations, return statements, and assignments *etc*. Figure 1 plots this classification, per project, for the top 5 categories. The results show that 30%-40% of one-line bug-fixes, the most dominant class, are stand-alone MI statements. And this does not include indirect MI changes, for example, changing an MI in the guard of an if condition, currently classified as an *if condition* change in Figure 1. Manually investigating each of the non-MI labeled bug-fixes revealed that almost 60% of *if condition changes*,

Bug Report Summary: *DateFormatUtils.format does not correctly change Calendar TimeZone in certain situations*

```
    public StringBuffer format(Calendar calendar, StringBuffer buf) {
        if (mTimeZoneForced) {
+           calendar.getTime(); /// LANG-538
            calendar = (Calendar) calendar.clone();
            calendar.setTimeZone(mTimeZone);
        }
        return applyRules(calendar, buf);
    }
```

Fig. 2. The bug report summary (top) and fix (bottom) for LANG-538

Bug Report Summary: *Field not initialized in constructor: org.apache.commons.lang.LocaleUtils.cAvailableLocaleSet*

```
    public static boolean isAvailableLocale(Locale locale) {
-       return cAvailableLocaleSet.contains(locale);
+       return availableLocaleList().contains(locale);
    }
```

Fig. 3. The bug report summary (top) and fix (bottom) for LANG-304

and at least 80% of other changes (variable initializations, assignments, and return expressions) involved MIs. In aggregate, 77% of the studied one-line bug-fixes involved MI changes, either stand-alone or part of another construct.

These results demonstrate the need to incorporate MI-related modifications, in a comprehensive manner, in the search space examined by a repair tool. The following examples illustrate the shortcomings of current tools in this respect.

B. Motivating Examples

Now we discuss two real-world examples that are beyond the scope of current repair tools, since they entail synthesis of substantially new MIs, typically rendered infeasible by a combinatorial explosion in the number of candidate patches.

Figure 2 presents a bug-fix (Bug ID: LANG-538) in the Apache Commons Lang project, taken from the popular Defects4J dataset. This patch requires the insertion of a new method invocation statement and is outside the repair space of current repair tools, since including such MIs would increase the repair search space significantly. For example, to fix this bug, ELIXIR synthesizes 836 valid MIs for that location (Table I). Certainly, validating such a large number of candidates, for a given repair location and a transformation schema, in a brute-force fashion is not practical. The only way the existing tools (such as SPR [8], PAR [12], or GenProg [3]) could attempt this bug fix would be by copying and pasting the same exact statement `calendar.getTime()` from elsewhere in the code. However, this statement is not present elsewhere.

Figure 3 shows another bug (LANG-304) also from Commons Lang in Defects4J. From an automatic repair point of view, this is also a non-trivial fix since the object of an MI is replaced by another MI that returns a compatible (type `List`) but not exactly the same type (type `Set`) of object. Current repair tools do not include such complex MI transformations in their repair space, to keep the search manageable. Also, the patch cannot be copied verbatim from elsewhere in the program either.

Proposed approach: Our proposed technique, ELIXIR can synthesize the correct patches for both the above bugs by, (1) first synthesizing a population of candidate patches and, (2) then ranking this candidate population and validating only the top several candidates. To synthesize the candidates ELIXIR first extracts atomic elements such as objects, variables, and literals in scope, and then synthesizes valid MI (*e.g.,* `obj.foo(a,b)`) and field access (*e.g.,* `obj.a`) expressions. These expressions constitute the building blocks (termed repair-expression) for synthesizing patches, and are then plugged into various program transformation schemas to generate candidate patches.

TABLE I
CANDIDATE PATCHES FOR LANG-538

Rank	Synthesized MIs
1	format(calendar)
2	calendar.clear()
3	format(calendar, buf)
:	
6	calendar.setLenient(mTimeZoneForced)
7	**calendar.getTime()**
:	Other 800+ Candidates

To effectively deal with the large set of candidate patches resulting from a rich set of repair-expressions, ELIXIR ranks the candidates using a machine-learned model and only selects the top few for validation against the test-suite. The machine-learned model is built on a set of four simple, but potent features, described in Section III-C. For our motivating example, bug LANG-538, when ELIXIR instantiates the MI-insertion schema (described in Section III-B1), the correct patch is ranked at 7^{th} out of 836 candidate patches (Table I) and hence can be quickly validated through the test-suite.

III. ELIXIR

The overall structure of ELIXIR is presented in Figure 4. For a given bug, ELIXIR takes as input a buggy program, a test suite (having at least one bug reproducing test case), and optionally a bug report, and produces a patch that passes all the test cases, *i.e.,* fixes the bug. ELIXIR works in four steps.

(A) Bug Localization. This step identifies a list of suspicious statements in the buggy program. Then for each potential buggy statement (repair location), ELIXIR performs the following steps until a *plausible patch*[1] is found.

(B) Generating Candidate Patches. ELIXIR includes a set of program transformation schemas described in Section III-B1. For each schema, ELIXIR generates a list of candidate patches by plugging in various repair-expressions into the schema, and performs the following steps until a plausible patch is found.

(C) Ranking and Selection of Candidate Patches. ELIXIR uses a machine-learned model to rank the candidate patches and selects the top N patches for validation.

(D) Validating Selected Candidate Patches. ELIXIR applies the selected patches one at a time, beginning from the top of the ranked-list, on the buggy program, and runs the test cases. If all the test cases pass, ELIXIR terminates and returns that patch as a plausible patch.

[1]A plausible patch is one that simply passes all test-cases (including failing tests) in the test suite, but may still be incorrect because the test-suite may provide an incomplete specification.

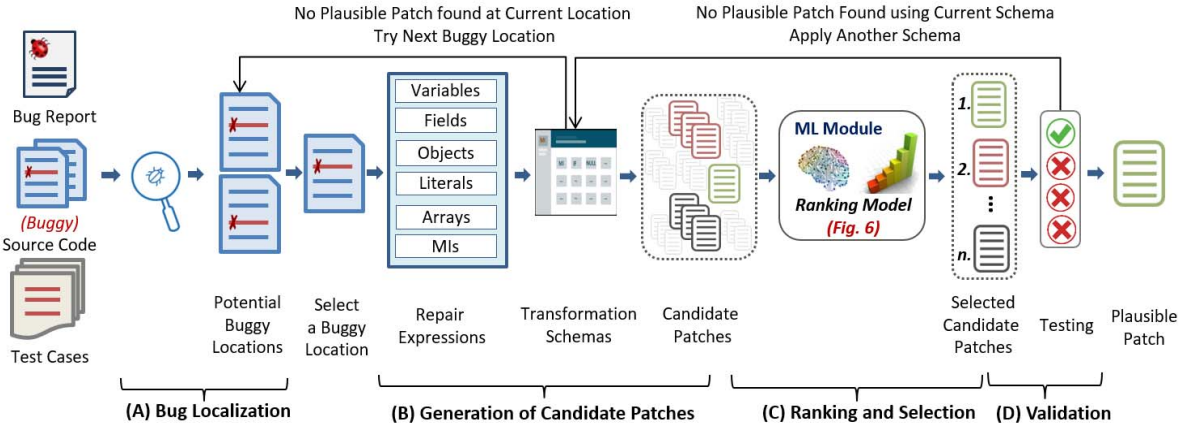

Fig. 4. Overview of ELIXIR.

A. Bug Localization

ELIXIR uses the Ochiai technique [27], a popular existing spectrum-based bug localization approach to identify potential buggy statements. According to the Ochiai technique, ELIXIR instruments the program at statement level, and collects test spectrum–*i.e.,* the statements executed by each test case, and computes a suspiciousness score for each statement. Finally, all statements are ranked in a descending order of supiciousness score, *i.e.,* the top statement is the most suspicious.

B. Generation of Candidate Patches

Fixing a bug involves applying appropriate changes at the buggy location. Allowing arbitrarily complex transformations can results in an infinite number of candidate patches. Therefore, program repair tools typically define their repair space through a fixed set of parameterized program transformation schemas, paired with a restricted set of expressions to instantiate those schemas. We term these expressions as *repair-expressions*. For example, the schema *Insertion of a Method Invocation* instantiated with the MI repair-expression `calendar.getTime()` yields the patch in Figure 2.

1) Program Transformation Schemas: ELIXIR applies the following program transformation schemas in the presented order to produce candidate patches for a given statement.

(T1) Widening Type: For a variable declaration statement, this schema replaces the type of the variable with a widened type – *e.g.,* `float` to `double`.

(T2) Changing Expression in Return Statement: This schema replaces a returned expression by another expression having compatible types.

(T3) Checking Null Pointer: If a statement has an object reference, this schema adds an *if guard* that ensures no null object is accessed.

(T4) Checking Array Range and Collection Size: If a statement has array references or collection objects, this schema adds an *if guard* to ensure that all array or collection accesses are within range, to prevent exceptions.

(T5) Changing Infix Boolean Operator: This schema includes common mutation operators from mutation testing research. For example, an infix expression like $a > b$ can be changed to $a \geq b$, $a < b$, and so on.

$$
\begin{aligned}
literal &\rightarrow \textbf{boolean} \mid \textbf{number} \mid \textbf{null} \\
varable &\rightarrow \textbf{id} \\
field &\rightarrow \textbf{id.id} \\
array &\rightarrow \textbf{id}[expression] \\
expression &\rightarrow literal \mid variable \mid field \mid array \\
argumentList &\rightarrow argumentList, expression \mid expression \\
methodInvocation &\rightarrow \textbf{id}(argumentList) \mid \textbf{id.id}(argumentList)
\end{aligned}
$$

Fig. 5. Grammar to Describe the Specific Repair-Expressions in ELIXIR. It should be that this grammar simply presents the structures of expressions. Please refer to the relevant documentation [28] for the accurate grammar.

(T6) Loosening and Tightening Boolean Expression : If a `boolean` expression is an if condition or in a return statement, this schema may remove or add predicates.

(T7) Changing Method Invocation (MI): This is a complex schema comprised of the following schemas:

- **Replacing Object Expression:** Replaces the object reference by another compatible-typed expression.
- **Replacing Method Name:** Replaces the method name with another method name having the same signature.
- **Replacing Argument:** Replaces an argument expression by another expression having compatible types.
- **Replacing a full MI by a synthesized MI:** Replaces the complete MI by a synthesized MI (can also be an overloaded MI) that returns a compatible type.

(T8) Insertion of a Method Invocation: This is a new schema in ELIXIR. It synthesizes MIs, and inserts them as a part of an expression or as a complete statement.

2) Synthesis of Repair-Expressions: One of the key features of ELIXIR is that it uses a rich set of repair-expressions that are effective in fixing real bugs in OO-programs. In particular, this entails a generous use of MIs and object accesses. Figure 5 shows the grammar of repair-expressions used in ELIXIR.

The synthesis of repair-expressions involves i) extracting relevant program elements in scope, and ii) creating repair-expressions using them. Specifically, given a repair location, ELIXIR extracts all the local variables and literals in scope, fields in the same class, and all the public fields in other classes that are relevant to the buggy class. Here, relevant classes are ones whose fields are accessed or methods are invoked within the buggy method. For example, for the bug

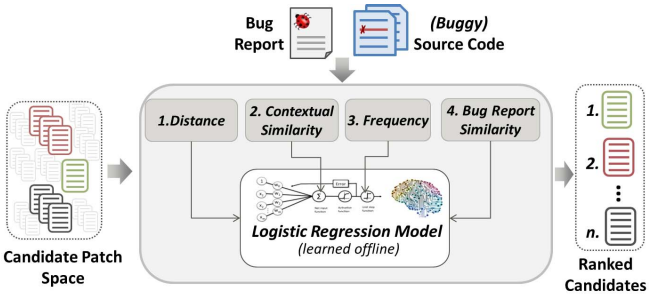

Fig. 6. Ranking and Selection of Repair-Expressions.

in Figure 2, `Calendar` and `StringBuffer` constitute relevant classes. Further, ELIXIR extracts the signatures of all the methods that can be invoked from the repair location. Next, ELIXIR creates field expressions (*e.g.,* `a.b`), and concrete MIs using the extracted method signatures, variables, literals, and fields, as per Figure 5. All these variables, literals, fields, and concrete method invocations constitute the pool of repair-expressions used in the next step.

3) Synthesis of Candidate Patches: The synthesized repair-expressions are used to instantiate the program transformation schemas to produce a set of concrete candidate patches. It should be noted that the first five schemas of ELIXIR (**T1** - **T5**) require only a small set of repair-expressions. For example, we have to only add null checkers for only those objects that are accessed in the repair location. Using widening type schema we can change `int a;` to {`long | float | double`} `a;`. Therefore, these schemas generate a small set of concrete patches. However, the rest of the schemas involve any repair-expressions defined in Figure 5. Therefore, they may produce a significantly large number of candidate patches. Although generation of patches is fast, compiling the program after applying a patch, and testing it is expensive. To cope with this search space explosion, ELIXIR ranks all the candidate patches and selects the top N patches for validation.

C. Ranking and Selection of Candidate Patches

Ranking candidate patches is a challenging problem since any valid (*i.e.,* compilable) patch can be the correct patch. Our key insight is that the program context and the bug report may provide valuable clues to identify the truly *relevant* patches. Therefore, we propose a machine learning technique to rank and select candidate patches. From a machine learning standpoint, the selection of repair candidates can be viewed as a binary classification problem, where our objective is to determine whether a particular candidate patch is relevant to a particular program context. Furthermore, we can compute a relevance score of each candidate patch, and use that to rank all the candidate patches. To this end, we use logistic regression, which is a widely used machine learning technique in practice.

Figure 6 presents the overall approach for ranking candidate patches. Given a set of candidate patches, ELIXIR first extracts four feature scores for each candidate patch from the used repair-expressions in it. These feature scores are calculated based on the repair context and the bug report. Then these feature scores are passed to an already trained logistic regression model that computes a probability score for each candidate patch representing its relevance. The learned logistic regression model is trained offline in advance using the same features from a set of previous bug fixes (training dataset). The subsequent sections describe the approach in more detail.

1) Selection of Feature Set and Calculation of Feature Scores: Based on an extensive study on prior literature on code completion, bug localization, and program repair techniques, we selected four features for our task: i) distance, ii) contextual similarity, iii) frequency in the context, and iv) bug report similarity. Note that repair-expressions used in a candidate patch is composed of one or more elements (e.g., variables/fields/literals). A variable itself is a single-element repair-expression, whereas an MI is multi-element repair-expression since several variables and objects may involve there. Therefore, we compute the feature scores of a patch in terms of the feature scores of new repair-expressions in the patch. For example, when we change a method parameter, the feature scores of the new parameter represent the patch feature scores. Therefore, ELIXIR first computes the feature score at element-level and then at patch-level.

Distance. Our first insight is that the more a repair-expression is composed of closer elements to the repair location, the more it is relevant. PAR [12] uses a similar concept in terms of AST nodes to sort the candidate variables. To compute the distance score of a repair-expression, ELIXIR first computes the distance score of each element (ξ) in the repair-expression from the repair location ($\Re$) using Equation 1.

$$\mathscr{S}_d = \begin{cases} 1 - \frac{ld(\xi,\Re)}{len(m)} & \text{if } len(m) \geq ld(\xi,\Re) \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

where $ld(\xi, \Re)$ is the number of comment-free lines between $\Re$ and the closest occurrence of ξ, and $len(m)$ is the number of comment-free lines in the method. For multi-element repair-expression, we average the distance scores of elements.

Contextual Similarity. Repair context, i.e., the surrounding code of repair location often provide useful hint to determine which repair-expressions are more consistent than others. Program context has been effectively used in auto code completion [29], API [20] and parameter recommendation [19]. Our insight is that the more a repair-expression is textually similar to the context, the more relevant it is for the repair location. We set the size of program context to six lines, i.e., three lines before and after the repair location. This size has been found to be effective in a recent work on API recommendation [20]. We compute the contextual similarity between a repair-expression and its program context as follows:

1) We extract the identifier names of the given repair-expression, and split CamelCase identifiers (`cAvailableLocaleSet`) into a set of tokens (`c`, `available`, `locale`, and `set`), which we call ($S1$).
2) We extract all the identifier names from the context. If the CamelCase repair-expression identifiers are exactly present in the context, we remove it. Otherwise, it would get similarity with itself. We split the resulting context identifiers, which becomes the context token set $S2$.

3) We compute the token similarity using Jaccard Similarity Coefficient (Equation 2)

$$\mathscr{S}_{context} = \frac{|S1 \cap S2|}{S1 \cup S2} \quad (2)$$

Frequency in the context. Our third insight is that the more a repair-expression is composed of frequently used elements, the more it is relevant in that context. However, it depends on the type of the element. For example, an object or a variable may be used repeatedly in a program context to perform some operations on it. However, an MI may not be used repeatedly in the same context. We performed a "Correlation-based Feature Subset Selection" technique from Weka toolkit [30] on our training dataset and found that frequency is not correlated with choosing method names but correlated with variables and objects. Therefore, for multi-element repair-expression we average the frequency of only objects and variables.

Bug Report Similarity. Bug reports have been widely used in information retrieval based bug localization [21], [22], [31]. The idea is based on the fact that bug reporters use similar words to describe a bug that are in the buggy source code. Sometimes, reporters also write about possible fixes. Liu et al. [23] used bug reports for fixing buffer overflows, null pointer bugs, and memory leaks in C programs. Our insight is that the information from the bug report can be used in prioritizing repair-expressions. To this end, we calculate the similarity score of a repair-expression with bug report in terms of token similarity using Equation 2, where $S1$ and $S2$ are the set of repair-expression and the bug report tokens respectively.

2) Logistic Regression Model for Ranking: The objective of our approach is to learn a model that aggregates the feature scores of a candidate patch ($\mathcal{P}$) in such a way that, for a given repair location, the candidate patch that are highly relevant would get a higher score than the irrelevant patches. Let us assume that for a given candidate patch, $\mathcal{P}$, the four feature scores: distance, bug report similarity, context similarity, and frequency are $\mathscr{S}_{dist}$, $\mathscr{S}_{con}$, $\mathscr{S}_{br}$, and $\mathscr{S}_{frq}$. ELIXIR aggregates the feature scores by a weighted sum:

$$f(\mathcal{P}, \theta) = \alpha \times \mathscr{S}_{dist} + \beta \times \mathscr{S}_{con} + \gamma \times \mathscr{S}_{br} + \zeta \times \mathscr{S}_{freq} \quad (3)$$

Here θ is the weight vector $[\alpha, \beta, \gamma, \zeta]$. These weights ($\theta$) are learned from a training dataset.

For a given patch $\mathscr{P}$ and its feature vector $[\mathscr{S}_{dist}, \mathscr{S}_{con}, \mathscr{S}_{br}, \mathscr{S}_{freq}]$, our objective is to compute the probability of $\mathcal{P}$ being relevant to the program context. Logistic regression machine learning technique is a powerful statistical way of modeling a binomial outcome with a probability score. Logistic regression function is defined as:

$$\sigma(t) = \frac{1}{1 + e^{-t}} \quad (4)$$

Substituting t in Equation 4 by $f(\mathcal{P}, \theta)$, we get:

$$\sigma(f(\mathcal{P}, \theta)) = \frac{1}{1 + e^{-f(\mathcal{P}, \theta)}} \quad (5)$$

For binomial classification, we can assume that a data instance follows Bernoulli distribution [32], which is:

$$p(f(\mathcal{P}, \theta), y|\theta) = \sigma(f(\mathcal{P}, \theta))^y (1 - \sigma(f(\mathcal{P}, \theta)))^{(1-y)} \quad (6)$$

TABLE II
DETAILS OF SUBJECTS IN DEFECTS4J [37]

Subject	#Bugs	KLOC	#Tests
Commons Math	106	85	3,602
Commons Lang	65	22	2,245
Joda-Time	27	28	4,130
JFreeChart	26	96	2,205

where y is 1 when the candidate patch is relevant and 0 when it is irrelevant. Therefore, to learn the weights of the model, θ, we compute $p(\theta|\xi, y)$, which is the posterior probability, from a training dataset. We use Weka's implementation of logistic regression [33] to learn θ.

IV. EXPERIMENTAL SETUP

A. Implementation

ELIXIR is implemented on the top of an automatic program repair framework, called FLAiR that we developed. FLAiR has its own bug localization system, various program transformation schemas, an in-memory compilation system, JUnit test case execution system, and a run-time data monitoring system. All the tools in FLAiR are written in Java, leveraging existing libraries where possible. More specifically, FLAIR bug localization system uses the ASM byte code library [34] to instrument programs' source code. FLAiR uses Spoon [35] library to modify a program at the abstract syntax tree (AST) level. After applying a repair schema, FLAIR leverages javax.tools [36] for in-memory compilation. Then FLAIR uses JUnit APIs to run the test cases programmatically. We also implement two other variants of ELIXIR on FLAiR framework.

ELIXIR-Baseline uses exactly the same program transformation schema as in ELIXIR. However, it uses the repair-expressions following the existing tools such as ACS, PAR, and HD-Repair. This baseline helps us to demonstrate the contribution of our rich set of repair-expressions.

ELIXIR-NoML uses both ELIXIR's schemas and repair-expressions. However, it selects N patches randomly instead of any machine learning technique. This baseline helps us to demonstrate the contribution of our proposed ranking model.

B. Dataset

In order to rigorously evaluate ELIXIR, we used two dataset.

1) Defects4J: Our first dataset is the popular Defects4J dataset [37]. We used the same four subjects from Defects4J that the existing repair tools are evaluated on. Table II (taken from [37]) presents the details of the dataset.

2) Bugs.jar: Our second dataset is Bugs.jar, created by us. The reasons for creating Bugs.jar are twofold: i) since ELIXIR is an ML-based approach, we need a training dataset, ii) evaluating ELIXIR on a different dataset than Defects4J.

Bugs.jar is a large-scale full-fledged real-world bug dataset. Since this is a new dataset, we briefly discuss our methodology to create Bugs.jar. Each bug in Bugs.jar contains (1) the buggy version of the source code, (2) a test-suite, serving as a correctness specification, comprising at least one failing (bug reproducing) test case and one passing test case (to guard against regression), and (3) the developer's patch to fix the

TABLE III
DETAILS OF BUGS.JAR

Project	Tags By Apache	Commits	Bugs Reports	Size [KLoC]	Bugs Selected
Accumulo	database	8,714	2,041	458	**98**
Camel	network-client/server	24,096	1,081	257	**147**
Commons Math	library	5,994	635	187	**147**
Flink	big data	8,906	2,070	345	**70**
Jackrabbit Oak	XML	10,810	1,686	228	**278**
Log4J2	library	6,971	784	104	**81**
Maven	build management	10,264	2,863	100	**48**
Wicket	web framework	19,386	3,770	177	**289**
Total		95,141	14,930	1,856	**1,158**

bug, which passes all the test cases. Furthermore, each bug in Bugs.jar has the original bug report associated with it.

Design Criteria. The design of our dataset was driven by four broad criteria: i) real-world relevance: having large, active projects, with a rich development history, ii) diversity: having projects covering the spectrum of applications, iii) reproducibility: having consistently reproducible bugs, and iv) automatability: building and testing the projects automatically.

Methodology. After conducting a rigorous search on GitHub and Google Code, we found that projects developed by the Apache Foundation fulfill our real-world relevance and automatability criteria. Further, there are several hundred projects in this ecosystem and projects are tagged with one or more of 28 different keywords, such as library, big data, etc., representing the application domain of the project. Thus we chose the Apache ecosystem on GitHub, which has 260 such tagged projects, for constructing our dataset. Then we selected the top 8 groups, each of which has at least 20 projects. This strategy respects our diversity criterion. Then for each group, we selected a representative subject that has at least 50KLoc and 5000 commits. Then for each project, we identified the bug fixing commits following Apache developers' convention that a Bug ID is present in the commit message. Then we consistently reproduce bugs by running test cases at least 10 times. Finally, we manually verified each reproducible bug. Table III provides the details of Bugs.jar.

C. Research Questions

We evaluate ELIXIR with respect to four research questions:

RQ1: How effective ELIXIR is compared to state-of-the-art on Defects4J?

RQ2: What is the contribution of repair-expressions, and ranking and selection model of ELIXIR?

RQ3: Are all features used in ELIXIR contributed toward the overall performance?

RQ4: Is ELIXIR's performance on Defects4J also reflected on Bugs.jar?

D. Training ELIXIR

For training ELIXIR, we used all the one-line bug-fixes from the subjects in Bugs.jar. For all these bugs, we extracted all the positive repair-expressions (that actually used in the repair) and all the negative repair-expressions with their feature vectors. Since negative repair-expressions are a lot more than the number of positive repair-expressions, to balance the training dataset, we replicated each positive repair-expressions 4 times, and for each positive repair-expression we randomly chose equal number of negative but similar (e.g., variables or MIs) repair-expressions. Similar strategy has been used in other work as well [32]. In this way, we obtained 1,580 data points, which is sufficient for our prediction model [38].

For evaluating ELIXIR on Defects4J, we used training dataset from all subjects except Commons Math in Bugs.jar. However, for evaluating ELIXIR on Bugs.jar, we removed the subject under evaluation from the training dataset. This makes sure that our training and testing dataset are always mutually exclusive. We followed the standard 10-fold cross validation methodology to train ELIXIR.

E. Evaluation Metric and Patch Correctness

We evaluate each tool in terms of number of correct and incorrect patches. We classify a patch as *correct*, if it is semantically equivalent to the developer-provided patch, based on a manual examination. This is consistent with previous work [39], [8], [40], [9], [10]. An *incorrect patch* is a patch that is not correct. To determine the correctness of a patch, two authors of the paper evaluated all the patches independently. In case of disagreement, we all had a group discussion until we had a mutual agreement.

F. Experimental Configurations

System. We used 2 Core of Intel(R) Core(TM) i7-4790 CPU of 3.60GHz and 4GB memory per instance for our experiment. We used Ubuntu 14.04 LTS operating system and Java 7.

ELIXIR. Currently, ELIXIR iterates through top 200 statements returned by the bug localization tool. For each schema, ELIXIR selects top 50 candidates returned by logistic regression model. ELIXIR's timeout is set to 90 minutes.

V. EXPERIMENTAL RESULTS

A. RQ1: Comparison with state-of-the-art approaches

To compare ELIXIR with state-of-the-art, we chose five state-of-the-art G&V repair tools: jGenProg [41], NOPOL[41], a reimplementation of PAR (we call PAR′) by Le et al. [14]), history driven repair (we call HD-Repair) [14], and ACS [15]. To the best of our knowledge, these include all the repair tools that were evaluated on Defects4J. Since automatic program repair experiments are very expensive, we discarded all the bugs that required multi-hunk fixes since they are by definition out of scope of ELIXIR. Similar strategy is also followed by Le et al. [14] while evaluating HD-Repair. Therefore, the presented results of ELIXIR for 82 bugs that required a single-hunk fix. The results of other tools are taken from the respective papers [41], [14], [15]. Table IV presents the results of each tool in terms of number of correct and incorrect patches. The overall results show that ELIXIR produced 26 *correct* patches, which is the highest on Defects4J. The second best is 18 patches by recently introduced ACS. All other tools generated 10 or less correct patches first. It should be noted that HD-Repair generated 16 patches but among them 10 were

TABLE IV
COMPARISON WITH EXISTING TECHNIQUES (CORRECT/INCORRECT)

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

NR=Not Reported.
* HD-Repair generated correct patches for 16 defects, but only 10 were ranked first [15]. All other tools terminate at the first plausible patch.

TABLE V
CONTRIBUTION OF PROGRAM TRANSFORMATION SCHEMAS (ELIXIR)

Transformation Schema	Correct	Incorrect
Change in MI	12	6
Change in Boolean Expression	6	8
Insertion of MI	3	0
Type Widening	2	0
Change in Return Expression	2	0
If Guard (Null/Array Size Checking)	1	1

ranked first. Since ELIXIR terminates at the first plausible patch, 10 is the fair number to compare for HD-Repair.

Since ACS fixed the second highest number patches, we further investigated the nature of patches by ACS and ELIXIR. We observed that only 4 patches are common between ELIXIR and ACS. This observation matches our expectation since ELIXIR targets more on MI-related bugs, whereas ACS targets condition-synthesis related bugs. It would be interesting to investigate how a tool performs that combines ELIXIR and ACS. However, that is beyond the scope of our current evaluation. Although bug specific results are not available for HD-Repair, from the results it is clear that ELIXIR (26 patches) fixed a lot more new bugs than HD-Repair (10 patches).

We also investigated which schemas of ELIXIR contributed more in generating the plausible (both correct and incorrect) patches. From Table V, we observe that changing MI in a comprehensive way is the most effective schema. It generated 12 correct patches, although it generated 6 incorrect patches. The second effective is the broad category of changes in Boolean expressions. This comprises several schema described in Section III-B1. They generated 6 correct patches. However, these schemas are also the source of many incorrect patches (8). Insertion of MI generated 3 correct patches with no incorrect patches. This result clearly demonstrates the effectiveness of ELIXIR on (prevalent) MI-related bugs.

B. RQ2: Contribution of ELIXIR*'s Repair-Expressions and Ranking and Selection*

From the results of RQ1, it is hard to understand the sole contribution of our rich repair-expressions since various tools targeted various kinds of bugs, used various repair-expressions and transformation schemas. Therefore, we ran our two baselines: ELIXIR-Baseline and ELIXIR-NoML on the same set of bugs in Defects4J that we used for RQ1. Recalling from Section IV-A, both baselines have the same program transformation schemas as of ELIXIR. ELIXIR-Baseline uses the repair-expressions following the existing tools, and ELIXIR-NoML uses ELIXIR's repair-expressions but selects 50 patches (same number as ELIXIR) randomly instead of any machine learning technique. We ran ELIXIR-Baseline 10 times due to its randomness, and counted the patches as correct even it generated the correct patches for one out of 10 times.

TABLE VI
CONTRIBUTION OF ELIXIR'S INGREDIENTS, AND RANKING AND SELECTION OF CANDIDATE PATCHES

Variant	Repair-Expressions	Ranking	Correct	Incorrect
ELIXIR-Baseline	Traditional	Off	14	16
ELIXIR-NoML	Extended	Random	13	5
ELIXIR	Extended	LR	26	15

LR = Logistic Regression

Table VI shows that even though we use the same set of transformation schemas of ELIXIR, ELIXIR-Baseline can fix 14 bugs. By comparing these results with Table IV, we see that ELIXIR-Baseline is almost as good as ACS, and outperforms other tools. However, ELIXIR-Baseline cannot generate any correct patches for the bugs that we presented in the motivating examples. Therefore, this result clearly demonstrates the contributions of our rich repair-expressions.

The results of ELIXIR-NoML demonstrates that simply extending the set of repair-expressions without any effective selection and pruning actually decreases the number of correct patches (14 vs. 13). In our experiments, we observed that median size of our expanded repair-expressions is 30 times larger than the basic repair-expressions. Certainly we cannot apply and validate all the patches resulting from them. Therefore, we picked 50 patches randomly. When we investigated each correct patch from ELIXIR-NoML, we observed that 10 patches are the same as that of ELIXIR-Baseline. Six of these 10 patches are not affected by the expanded repair-expressions because the used schemas (T1-T5 in Section III-B1) are repair-expressions independent. Due to expanded repair-expressions, although ELIXIR-NoML generated correct patches for 3 new bugs that ELIXIR-Baseline could not produce, it lost patches for 4 bugs that ELIXIR-Baseline produced. However, with the machine learning technique, ELIXIR has not lost any patches that ELIXIR-Baseline generated, and additionally it generated correct patches for 12 more bugs.

C. RQ3: Effect of Each Feature in Ranking and Selection

To investigate whether all the features contributed in the ranking and selection of candidate patches, we turned off one of the four features at a time during the training and testing phase of ELIXIR. The results in Table VII indeed show that each feature contributed in the ranking of correct patch in the search space. Turning off any feature reduces the number of correct patches. Among them distance turned out to be the least influential feature whereas the bug report and frequency are the dominant features. More specifically, ELIXIR lost only one patch in absence of distance, whereas lost 5 and 6 patches in absence of frequency and bug report respectively.

Figure 7 presents an example where ELIXIR could not generate the correct patch when we turned off the frequency fea-

TABLE VII
EFFECT OF EACH FEATURE IN RANKING AND SELECTION

Features	Correct	Incorrect
All-Distance	25	15
All-Context	23	15
All-Bug Report	20	19
All-Frequency	21	17

```
   public static float max(final float a, final float b) {
-      return (a <= b) ? b : (Float.isNaN(a+b) ? Float.NaN : b);
+      return (a <= b) ? b : (Float.isNaN(a+b) ? Float.NaN : a);
    }
```

Fig. 7. Fix of MATH-482

ture. From the bug-fix we can see that repair location returns a very complicated expression which is basically a ternary `if` expression. Here the mistake is that `b` is returned instead of `a`. This repair location generates a huge number of candidate patches since it has MI, return statement etc. In fact, each of `a`, `b`, and `Float.NAN` can be replaced by 234 repair-expressions. Some of the candidates include `Float.POSITIVE_INFINITY`, `Float.MIN_VALUE`, `min(a, b)` etc. However, since `a` is one of the most frequent variables in the context, it got a high score and ELIXIR generated the correct patch.

The example in Figure 8 shows the effect of bug report similarity feature. ELIXIR synthesizes 944 valid MIs for the repair location. From the bug-fix, we see that the inserted MI does not have any element from the context. However, since the bug report contains both the terms `next` and `pos`, ELIXIR ranks it at 3rd position. Due to lack of space, we do not provide concrete examples for distance and context features.

D. RQ4: Effectiveness of ELIXIR on Bugs.jar

Finally, we ran ELIXIR on Bugs.jar to understand its effectiveness on Bugs.jar w.r.t. ELIXIR-Baseline. Since Bugs.jar has a large number of bugs, we randomly sampled 50% of one-hunk bugs for this experiment. We could not run our tools on Log4J2 due to some engineering issues. More specifically, our framework and many of its dependent libraries used Log4J2 for logging, which was interfering with the "subject Log4J2". This is an interesting engineering problem to solve in the future. Therefore, we randomly picked 127 bugs from 7 subjects in Bugs.jar to create our sample set.

From Table VIII, we observe that ELIXIR generated correct patches for 22 bugs, whereas ELIXIR-Baseline generated a correct patch for 14 bugs. The 8 new correct patches generated by ELIXIR involved 4 MI insertions, 2 MI changes, 2 fields. It should be noted that in RQ1, we demonstrated that ELIXIR-Baseline is better than state-of-the-art techniques except ACS. Therefore, the results in Table VIII demonstrate that ELIXIR is similarly effective on Bugs.jar compared to the state-of-the-art. The similarity in improvement also demonstrates that *Bugs.jar is not biased toward* ELIXIR.

When we investigated the correct patches by ELIXIR, we found many small but complicated fixes. Figure 9 presents a concrete example that shows the fix for `CAMEL-7241`, where ELIXIR replaced a parameter of an MI, which is also an MI by a completely different MI that ELIXIR synthesized.

```
    if (escapingOn & cstart == QUOTE) {
+       next(pos);
        return appendTo == null ? null : appendTo.append(QUOTE);
    }
```

Fig. 8. Fix of LANG-477

TABLE VIII
PATCH GENERATION SUMMARY (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

VI. THREATS TO VALIDITY

External validity. Our evaluation is conducted only on the Defects4J and Bugs.jar datasets and our conclusions may not generalize to subject systems and bugs beyond these datasets. The use of two, independently constructed datasets was in part a conscious choice to mitigate this threat. Further, in constructing Bugs.jar we followed a rigorous, scientific procedure to identify 8 diverse, representative subject systems, and included *all* their reproducible, single-module bugs in our dataset. Second, ELIXIR has currently only been instantiated and validated on Java application bugs. While the ELIXIR technique is conceptually general, the current results may not generalize to bugs in other OO-languages, such as C++.

Internal validity. Any implementation error of ELIXIR could impact internal validity. We mitigated this by manual inspection of all patches produced by the tools and by following good development and QA practices.

The Bugs.jar dataset and hence our use of it may pose several threats too. In creating Bugs.jar, we followed a rigorous and scientific procedure for selecting subject systems. Therefore, Bugs.jar is not biased toward any specific type of bugs or subjects that is beneficial to ELIXIR. Further, errors in the classification of commits as bugs or feature-enhancements, due to mis-labeling in JIRA, or errors in our bug-extraction scripts, could pose a threat. We mitigated this threat by manually re-examining each selected bug to ensure it was indeed a bug. Flaky bugs, which may produce unpredictable behavior during repair, also pose a threat, which we mitigated by selecting only bugs that consistently re-produced in 10 runs.

Finally, ELIXIR-NoML, which uses randomized prioritization of candidate patches, is run 10 times to generate a patch.

Construct validity. Our criterion for classifying patches as *correct* or *incorrect* is based on manual analysis, which is not scientifically rigorous, even though it is accepted practice in previous work [39], [8], [40], [10]. We tried to mitigate this threat by following the protocol in Section IV-E, which involved independent classification of the patches by the first two authors and reconciliation of any discrepancies through deeper examination of the patch in question by all authors.

```
public static String toString(ByteBuffer buffer,
             Exchange exchange) throws IOException {
-   return IOConverter.toString(buffer.array(),exchange);
+   return IOConverter.toString(toByteArray(buffer),exchange);
}
```

Fig. 9. Fix of CAMEL-7241

VII. Related Work

Search-based repair for C programs. GenProg [3], which pioneered this area, uses genetic programming to search a space of repair mutations formed by code snippets copied from elsewhere in the program. RSRepair [5] uses random search instead, while AE [4] uses deterministic search coupled by analysis to prune equivalent patches. Relifix [42] proposes a set of specialized repair schemas customized for repairing software regression errors. SPR [8] prioritizes repair of conditional statements, using abstract repair conditions to implicitly evaluate and prune away infeasible condition repair candidates (staging) before generating concrete repairs. In recent work, Tan et al. [11] propose the use of *anti-patterns* – a set of generic forbidden repair transformations, to reduce the incidence of plausible patches, that typically arise in search-based program repair, due to weak test-suites. The above contributions can, and in part have, been re-purposed for repair of Java (or other OO) programs. Our proposed technique seeks to substantially expand the repair space, by using richer repair expressions incorporating method invocations, and by efficiently searching this space using a machine-learnt model. Thus, it nicely complements the above body of work.

Search-based repair for Java programs. PAR [12] overlays GenProg's search strategy with a set of repair templates manually derived from human-written patches. History-driven repair [14] uses a rich set of templates drawn from GenProg, PAR, and mutation testing, to produce a large pool of candidate repairs which it then prioritizes and prunes based on the frequency of previous (human-written) patches. In a very recent work, ACS [15] proposes a method for precise condition synthesis by instantiating variables in predicates that frequently occur in a given corpus of code, using various heuristics to rank and choose the variables. While each of these techniques handles the Java language as such, unlike us, they use method invocations in very limited and specific ways, ostensibly to avoid an explosion in the repair search space. By contrast, the expanded and generalized use of method invocations in repairs is the main contribution of our work.

Oracle-based repair. SemFix [6] uses symbolic execution to create an oracular representation of an expression under repair and then uses program synthesis to generate a repaired statement compatible with this "oracle". MintHint [43] follows SemFix in creating the oracle but uses statistical analysis to search for a repair. DirectFix [7] generates *minimal* repairs to obtain comprehensible repairs by encoding the problem as a partial maximum satisfiability problem over SMT formulas. Angelix [10] solves the scalability problems of DirectFix by using a lightweight repair constraint. SearchRepair [44] uses *semantic search* to search a corpus of human-written patches, encoded as satisfiability modulo theories (SMT) constraints, for possible matches to a repair problem. All of these techniques target C programs. The only exception is NOPOL [13], which targets Java programs. It focuses on the repair of branch conditions, using instrumented test-suite executions to synthesize an oracle, which is converted into a suitable SMT formula and solved to obtain a patch. Like search-based Java repair techniques, NOPOL also incorporates method invocations in a very limited way to curb explosion of the repair space. In principle, our repair space ranking ideas could be applied in the patch-synthesis stage of an oracle-based repair technique, *i.e.,* in generation of a concrete patch from the oracle. This could constitute interesting future work.

AI in program repair. This is a nascent branch of research in program repair. Prophet [9] builds on the SPR technique, further using a machine learned model of previously-known correct human patches to prioritize candidate repairs. Conceptually, ELIXIR also uses machine learning to rank repair candidates. However, unlike Prophet's elaborate model with over 3000 features, which produces only a 25% improvement in repair outcomes (from 12 to 15 patches) we use a simple model with only 4 potent features, interestingly with substantially better repair outcomes (85% improvement from 14 to 26 patches). Also, our use of machine learning is paired with a meaningful expansion in the repair space to target a specific aspect of patches, *i.e.,* incorporating MIs in repair expressions. DeepFix [45] employs deep learning to fix language-level common programming errors in C programs. This work while interesting in its own right, is somewhat different from the test-suite based repair of functional errors, targeted by vast majority of program repair research, including ours.

VIII. Conclusions

This work was motivated by the extensive use of method invocations (MI) in object-oriented (OO) programs, and indeed their prevalence in patches of OO-program bugs. We proposed a generate-and-validate repair technique, ELIXIR, designed for repair of OO-programs, and instantiated it for Java program repair. ELIXIR aggressively uses MIs, 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-learnt model to rank concrete repairs. The model relies on four features derived from the program context, *i.e.,* the code surrounding the repair location, and from the bug report. ELIXIR was evaluated on two separate datasets, namely the popular Defects4J dataset, and another large-scale dataset Bugs.jar created by us. The evaluation shows that, by enlarging and effectively searching the larger repair space, ELIXIR is able to significantly increase the number of correctly repaired bugs while also out-performing other state-of-the-art tools like ACS, HD-Repair, NOPOL, PAR, and jGenProg.

We believe that this work is a promising demonstration of how AI/ML techniques can be used to expand the scope of automatic program repair techniques. In future, we hope to pursue and realize the full potential of this branch of research.

REFERENCES

[1] C. University, "Cambridge university study states software bugs cost economy $312 billion per year," http://www.prweb.com/releases/2013/1/prweb10298185.htm, 2013.

[2] T. Software, "Tiobe index," http://www.tiobe.com/tiobe-index/, May 2017.

[3] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 3–13.

[4] W. Weimer, Z. P. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. Piscataway, NJ, USA: IEEE Press, Nov 2013, pp. 356–366.

[5] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 254–265.

[6] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "SemFix: Program repair via semantic analysis," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 772–781.

[7] S. Mechtaev, J. Yi, and A. Roychoudhury, "DirectFix: Looking for simple program repairs," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 448–458.

[8] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 166–178.

[9] ——, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '16. New York, NY, USA: ACM, 2016, pp. 298–312.

[10] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 691–701.

[11] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury, "Anti-patterns in search-based program repair," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 727–738.

[12] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 802–811.

[13] J. Xuan, M. Martinez, F. DeMarco, M. Clment, S. L. Marcote, T. Durieux, D. L. Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, Jan 2017.

[14] X. B. D. Le, D. Lo, and C. L. Goues, "History driven program repair," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. Piscataway, NJ, USA: IEEE Press, March 2016, pp. 213–224.

[15] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," 2017, to appear.

[16] S. N. Ahsan, J. Ferzund, and F. Wotawa, "Are there language specific bug patterns? results obtained from a case study using mozilla," in *Software Engineering Advances, 2009. ICSEA '09. Fourth International Conference on*, Sept 2009, pp. 210–215.

[17] G. Booch, *Object-Oriented Analysis and Design with Applications (3rd Edition)*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2004.

[18] M. Martinez and M. Monperrus, "Mining software repair models for reasoning on the search space of automated program fixing," *CoRR*, vol. abs/1311.3414, 2013. [Online]. Available: http://arxiv.org/abs/1311.3414

[19] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou, "Cscc: Simple, efficient, context sensitive code completion," in *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ser. ICSME '14. IEEE Computer Society, 2014, pp. 71–80.

[20] A. T. Nguyen, M. Hilton, M. Codoban, H. A. Nguyen, L. Mast, E. Rademacher, T. N. Nguyen, and D. Dig, "Api code recommendation using statistical learning from fine-grained changes," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. ACM, 2016, pp. 511–522.

[21] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed?-more accurate information retrieval-based bug localization based on bug reports," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 14–24.

[22] X. Ye, R. Bunescu, and C. Liu, "Learning to rank relevant files for bug reports using domain knowledge," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 689–699.

[23] C. Liu, J. Yang, L. Tan, and M. Hafiz, "R2fix: Automatically generating bug fixes from bug reports," in *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*. IEEE, 2013, pp. 282–291.

[24] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "Bugs.jar: A large-scale, diverse dataset of bugs for java program repair," https://github.com/bugs-dot-jar/bugs-dot-jar, 2017.

[25] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, "The manybugs and introclass benchmarks for automated repair of c programs," *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1236–1256, 2015.

[26] B. Fluri, M. Wuersch, M. PInzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Trans. Softw. Eng.*, vol. 33, no. 11, pp. 725–743, Nov. 2007.

[27] R. Abreu, P. Zoeteweij, and A. J. C. v. Gemund, "An evaluation of similarity coefficients for software fault localization," in *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, ser. PRDC '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 39–46.

[28] "The java language specification," https://docs.oracle.com/javase/specs/jls/se7/html/index.html, 2013.

[29] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. IEEE Press, 2012, pp. 837–847.

[30] M. A. Hall, "Correlation-based feature subset selection for machine learning," Ph.D. dissertation, University of Waikato, Hamilton, New Zealand, 1998.

[31] X. Ye, H. Shen, X. Ma, R. Bunescu, and C. Liu, "From word embeddings to document similarities for improved information retrieval in software engineering," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. ACM, 2016, pp. 404–415.

[32] T.-D. B. Le, R. J. Oentaryo, and D. Lo, "Information retrieval and spectrum based bug localization: better together," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 579–590.

[33] S. le Cessie and J. van Houwelingen, "Ridge estimators in logistic regression," *Applied Statistics*, vol. 41, no. 1, pp. 191–201, 1992.

[34] ASM, "ASM," http://asm.ow2.org/.

[35] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "Spoon: A library for implementing analyses and transformations of java source code," *Software: Practice and Experience*, vol. 46, pp. 1155–1179, 2015. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01078532/document

[36] Java, "Javax Tools," https://docs.oracle.com/javase/7/docs/api/javax/tools/package-summary.html.

[37] R. Just, D. Jalali, and M. D. Ernst, "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*. ACM, 2014, pp. 437–440.

[38] G. Booch, *Essentials of behavioral research: Methods and data analysis (Volume 2)*. New York: McGraw-Hill, 1991.

[39] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: ACM, 2015, pp. 24–36.

[40] T. Durieux, M. Martinez, M. Monperrus, R. Sommerard, and J. Xuan, "Automatic repair of real bugs: An experience report on the defects4j dataset," *CoRR*, vol. abs/1505.07002, 2015. [Online]. Available: http://arxiv.org/abs/1505.07002

[41] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, "Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset," *Empirical Software Engineering*, pp. 1–29, 2016.

[42] S. H. Tan and A. Roychoudhury, "Relifix: Automated repair of software regressions," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 471–482.

[43] S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso, "Minthint: Automated synthesis of repair hints," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 266–276.

[44] Y. Ke, K. T. Stolee, C. L. Goues, and Y. Brun, "Repairing programs with semantic code search (t)," in *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ser. ASE '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 295–306.

[45] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "DeepFix: Fixing common c language errors by deep learning," in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*. AAAI Press, 2017.