The Journal of Systems & Software 188 (2022) 111263

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Estimating the potential of program repair search spaces with commit analysis ☆

Khashayar Etemadi [a,*], Niloofar Tarighat [b], Siddharth Yadav [c], Matias Martinez [d], Martin Monperrus [a]

[a] KTH Royal Institute of Technology, Sweden
[b] Sharif University of Technology, Iran
[c] Indraprastha Institute of Information Technology Delhi, India
[d] Université Polytechnique Hauts-de-France, France

ARTICLE INFO

Article history:
Received 26 May 2021
Received in revised form 27 January 2022
Accepted 3 February 2022
Available online 11 February 2022

Keywords:
Program repair
Search-space
Static code analysis
Commit analysis

ABSTRACT

The most natural method for evaluating program repair systems is to run them on bug datasets, such as Defects4J. Yet, using this evaluation technique on arbitrary real-world programs requires heavy configuration. In this paper, we propose a purely static method to evaluate the potential of the search space of repair approaches. This new method enables researchers and practitioners to encode the search spaces of repair approaches and select potentially useful ones without struggling with tool configuration and execution. We encode the search spaces by specifying the repair strategies they employ. Next, we use the specifications to check whether past commits lie in repair search spaces. For a repair approach, including many human-written past commits in its search space indicates its potential to generate useful patches. We implement our evaluation method in LIGHTER. LIGHTER gets a Git repository and outputs a list of commits whose source code changes lie in repair search spaces. We run LIGHTER on 55,309 commits from the history of 72 Github repositories with and show that LIGHTER's precision and recall are 77% and 92%, respectively. Overall, our experiments show that our novel method is both lightweight and effective to study the search space of program repair approaches.

1. Introduction

Fixing software bugs is a notoriously time-consuming task for developers (Murphy-Hill et al., 2013). To address this issue, automatic program repair (APR) approaches apply repair strategies to fix software bugs without human intervention (Monperrus, 2017). Researchers usually assess repair approaches by running them on bug datasets, such as Defects4J (Just et al., 2014), Bugs.jar (Saha et al., 2018) and ManyBugs (Goues et al., 2017). Comparative evaluations of repair systems (e.g., Liu et al. (2020), Martinez et al. (2017), Durieux et al. (2019)) have shown promising results in terms of the number of bugs that can be fixed in a given dataset.

Even though executing repair approaches is the most natural method for evaluating APR, there are two main obstacles when this evaluation is done on an arbitrary software project. First, fully executing a repair approach on a real world project often requires heavy and time-consuming configuration of the repair approach and the target project (Durieux et al., 2019). Second, the target programs under repair should have a test suite specifying their correct behavior and, at least, one failing test case that exposes the bug. Previous work (Tufano et al., 2017; Madeiral et al., 2019) showed it is hard to find real world commits with test suites that can be compiled and executed. These two major obstacles (configuration and dependability on strong testing) hinder assessment of automated program repair on new projects.

In this paper, we propose a new lightweight method to check whether repair approaches may be fruitful for new projects. Instead of fully executing a repair system to check if it actually fixes certain bugs, we analyze whether real world bug-fixing commits lie in the considered repair search space (Martinez and Monperrus, 2015). In this context, the search space of the repair approach is the set of all program patches that can be potentially generated. For example, GenProg (Le Goues et al., 2012a)'s search space contains all replacements of statements with a new one, where the new statement is copied from the program under repair. In this work, we first specify search spaces of repair approaches based on code patterns. This enables us to then compare real world commits against our specifications of a repair approach

☆ Editor: Alexander Serebrenik.
* Corresponding author.
E-mail addresses: khaes@kth.se (K. Etemadi), tarighat_niloofar@ee.sharif.edu (N. Tarighat), siddharth16268@iiitd.ac.in (S. Yadav), matias.martinez@uphf.fr (M. Martinez), monperrus@kth.se (M. Monperrus).

https://doi.org/10.1016/j.jss.2022.111263

search space as follows. For each commit, we check if it lies in the search space of a known repair approach. If that happens, we say that the approach covers that commit, and vice versa, that the commit could have been potentially produced by the repair approach. A repair approach that covers many real world human-written commits is likely to create useful patches in the future. In other words, the higher the commit coverage, the broader the search space of the approach.

We implement our novel method in LighteR. For a given commit that changes a program, LighteR performs static analysis on the abstract syntax tree (AST) to determine if the commit is covered by certain repair approaches. LighteR specifies the search spaces of 8 notable repair systems: Arja (Yuan and Banzhaf, 2018), Cardumen (Martinez and Monperrus, 2018), Elixir (Saha et al., 2017), GenProg (Le Goues et al., 2012a), jMutRepair (Martinez and Monperrus, 2016), Kali (Qi et al., 2015), Nopol (Xuan et al., 2016), and NPEfix (Cornu et al., 2015).

We run LighteR on 55,309 commits of 72 projects from Bears (Madeiral et al., 2019) to study the effectiveness of LighteR. Our experiments show that 747/55,309 of all the considered commits lie in the search space of at least one repair system. We also demonstrate that there is little overlap between the repair systems, showing that program repair research is producing systems that are complementary in practice. The median time LighteR spends to check a commit against the search space of a repair approach is 0.81 s, which is fast enough for practical usage. In another experiment, we measure how accurately LighteR determines whether a commit lies in the search space of a repair system compared to a ground-truth classification. Our results show that LighteR has a precision and recall of 77% and 92% respectively. Overall LighteR is useful to estimate the potential of the search space of repair approaches on a new project, without going through full configuration and execution of actual tools.

Few studies have analyzed the search spaces of repair approaches (Martinez and Monperrus, 2015; Long and Rinard, 2016a), and all of them with a different purpose than ours. We are the first to propose a lightweight method for conducting a fast evaluation of the breadth of the search space of repair approaches.

To sum up, our contributions are:

- A novel method for specifying the search space of program repair approaches, appropriate to study the potential of program repair to create patches corresponding to past commits of software repositories. This framework implemented in a tool called LighteR, is lightweight, it does not require configuration and execution of repair systems.
- A comprehensive series of experiments on past commits. By analyzing 55,309 human-written commits from 72 Github repositories, we show that 1.35% (747/55, 309) of past commits lie in the search space of at least one of the considered repair systems, 62% of these commits are indeed bug-fixing commits according to the manual inspection we conducted. Overall, our experiments show that our novel method is an effective means to study the potential of program repair search space.
- A systematic measurement of the reliability of LighteR. Our prototype system has a precision and recall of 77% and 92%, respectively, which is arguably high compared to close tools, such as PPD (Madeiral et al., 2018).

The rest of this paper is organized as follows: Section 2 presents the terminology that we use in this paper. Section 3 presents the different types of evaluation in program repair. Section 4 describes our proposed method. Section 5 and Section 6 explain the methodology and then the results of our experiments. Implications of our results are discussed in Section 7. Section 8 reviews the related work. Finally, Section 9 concludes this paper.

```
 Set<E> removed = getAllEdges(sourceVertex, targetVertex);
+ if(removed == null) {
+    return null;
+ }
 removeAllEdges(removed);
```

Listing 1: Commit 275c6fdb in Jgrapht, which is in the search space of NPEFix, it applies repair strategy "skip method".

2. Terminology

We use the following concepts throughout this study:

Automatic Repair Approach: A software artifact that gets a buggy version of a program as input and generates patches that fix the bug as output (Weimer et al., 2009). To generate the patches, a repair approach also requires an oracle that determines whether a version of a program is buggy or correct. For example, *test-suite based program repair approaches* use test-suites as the oracle (Le Goues et al., 2012a).

Repair Operator: A type of atomic change that is applied on the buggy program to repair the bug. For example, removing a statement from the source code is an operator used by Kali (Qi et al., 2015).

Repair Strategy: A set of repair operators applied in conjunction by a repair approach to the buggy version of a program. For example, one of the strategies employed by NPEfix (Cornu et al., 2015) is "skip method" (e.g., see Listing 1). Per this strategy, an if-statement is added before a suspicious statement. The corresponding if-condition checks whether a variable used by the suspicious statement is equal to null. If the if-condition holds, a return statement is executed.

Repair Ingredient: An existing source code fragment that is reused by a repair approach to fix the bug (Martinez et al., 2014; White et al., 2019). For example, in one of its repair strategies, GenProg (Le Goues et al., 2012a) creates a candidate patch by replacing a suspicious statement by another existing statement written elsewhere in the program. The latter is the *ingredient* of the candidate patch. Note that ingredients can have different granularities. For example, in GenProg, an ingredient is a statement, in NPEfix (Cornu et al., 2015) it is a variable, and in Cardumen (Martinez and Monperrus, 2018) it is an expression.

Scope of Ingredients: The scope of ingredients is the parts of program that are considered for extracting repair ingredients (Martinez et al., 2014; White et al., 2019). For example, jGenProg (Martinez and Monperrus, 2016) can replace an old statement s (written in file f from package p) with a new one, according to three different scopes: (1) same file (i.e., f), (2) same package (i.e., from any file belonging to p), and (3) same program.

Search Space of Repair Approach: Let us assume a repair approach r with certain repair strategies and a scope of ingredients. When a program is given as the input, the search space of r is the set of all patches that r can generate given the strategies and scope of ingredients (Martinez and Monperrus, 2015).

3. Types of evaluation in program repair

There are various ways for evaluating program repair approaches. In this section, we classify these techniques into two categories, dynamic evaluation and static evaluation, and we discuss their use cases and limitations.

Table 1
Execution steps of program repair approaches and example studies that focus on them in isolation.

Step	Kind	Focus	This Paper
Assumption Verification	Static	Madeiral et al. (2019)	
Fault Localization	Dynamic	Liu et al. (2019a)	
Ingredient Extraction	Static	Martinez et al. (2014), Barr et al. (2014)	✓
Code Synthesis	Static	Martinez and Monperrus (2015)	✓
Test Validation	Dynamic	Le et al. (2018)	
Overfitting Detection	Dynamic/Static	Long and Rinard (2016a), Le et al. (2016)	

3.1. Program repair steps considered in scientific evaluation

Assumption Verification: Test-based repair approaches assume the presence of a failing test that exposes the bug. Similarly, the repair system should be able to successfully build the program under repair before fixing it. These are core assumptions of test-suite based repair. Bug datasets facilitate program repair research by curating the buggy programs that meet those repair assumptions (Madeiral et al., 2019). We note that many bugs and their fixes exist in repositories without satisfying those assumptions, yet providing valuable knowledge for program repair research.

Fault Localization: This step refers to the process of ranking locations in the buggy program based on their likelihood to cause a bug (Wong et al., 2016). Repair approaches take advantage of fault localization methods to find the best candidate locations that should be changed to fix a bug. It is possible to isolate fault localization in program repair to study its importance (Liu et al., 2019a).

Ingredient Extraction: Redundancy based program repair approaches have an "ingredient extraction" step. In this step, the repair system extracts code components in the existing program that may be used for patch generation. This step is usually performed statically. Researchers have studied this step in isolation (Martinez et al., 2014; Barr et al., 2014).

Code Synthesis: A program repair patch is composed of code that is synthesized, possibly from ingredients in the case of redundancy based repair (Le Goues et al., 2012a). To do this, templates and code transformations are applied on the AST of the program. This step is static in most of the related work, with the exception of the dynamic collection of ingredients in Durieux and Monperrus (2016). It results in a set of candidate patches. An example of a study of code synthesis which is purely static is by Martinez and Monperrus (2015)

Test Validation: This is the step where all the tests are executed on the candidate patches, in order to discard the incorrect patches that do not pass them. An example of a study dedicated to this step is Le et al. (2018).

Overfitting Detection: The patches that pass all the tests but introduce regressions are filtered out based on static (Ye et al., 2021b) or dynamic analysis (Xiong et al., 2018). This is called the overfitting detection step. This has been studied in isolation for example in Long and Rinard (2016a).

Table 1 summarizes those different steps of program repair. The "Kind" column shows if the corresponding step of repair is carried out statically or if it requires execution of the program under repair. The "Focus" column shows an example of studies that specifically evaluate a given step. Finally, the last column indicates whether the corresponding step is considered in our novel evaluation technique proposed in this paper.

3.2. Types of evaluations

Now, it is clear that we classify the evaluations in program repair into two main groups: dynamic evaluation and static evaluation.

Dynamic evaluation methods focus on the dynamic steps of the repair process, and typically consist of running actual repair tools. By running the actual repair tools, these evaluation techniques may produce actual patches generated by the tools. For example, the RepairThemAll study (Durieux et al., 2019) executed 11 repair tools over five benchmarks, this is a archetypal dynamic evaluation. Evaluations of this type heavily depend on the feasibility of execution. For this reason, they require fine-tuned bug datasets appropriate for running repair tools on them, such as Defects4J (Just et al., 2014). Because of this big curation effort, only a few datasets have been created accordingly. This leads to repair tools over-engineered to fix specific bugs in those datasets, which in turn causes overestimation of the generalizability (Durieux et al., 2019). The main advantage of dynamic evaluations is that it give concrete insights for practitioners. The main limitation is that it is very costly, and thus tends to be limited to the same bugs or benchmarks again and again.

Static evaluation methods focus on static studying some steps of the repair process, namely, ingredient extraction, code synthesis, and static overfitting detection. Static evaluation does not require collecting dynamic execution data for repair approaches, hence it can be performed without running the actual repair tools. For example, Martinez and Monperrus (2015) study the frequency of applying repair operators in a large dataset of human-made commits. This means they only consider the code synthesis step, which can be evaluated by a static analysis of target commits. The main advantage of static evaluation is that it saves engineering resources and experimental time (for environment setup, configuration, and execution) and it does not require presence of extensive test suites. The main limitation is its abstractness, it does not tell which candidate patches will be delivered by actual repair tools.

In this paper, we fully concentrate on repair strategies and ingredients. Thus, it fits perfectly with static evaluation. We propose a conceptual framework and its implementation in LIGHTER to statically evaluate the search space of repair approaches and their strategies. LIGHTER fully focuses on the ingredient extraction and the code synthesis step of the repair process, which are both amenable to static study. In other words, we statically investigate the potential of repair approaches in terms of the breadth of their search spaces..

4. A lightweight method for analyzing program repair search spaces

4.1. Overview

The goal of our proposed method is to evaluate and compare different repair approaches in terms of the number of human-written patches that lie in their search space. For this purpose, we specify the search spaces of well-known repair approaches, and analyze the past commits of open-source repositories to compute the *commit coverage* of each repair approach, as follows. To define commit coverage, we first define repair-space commit.

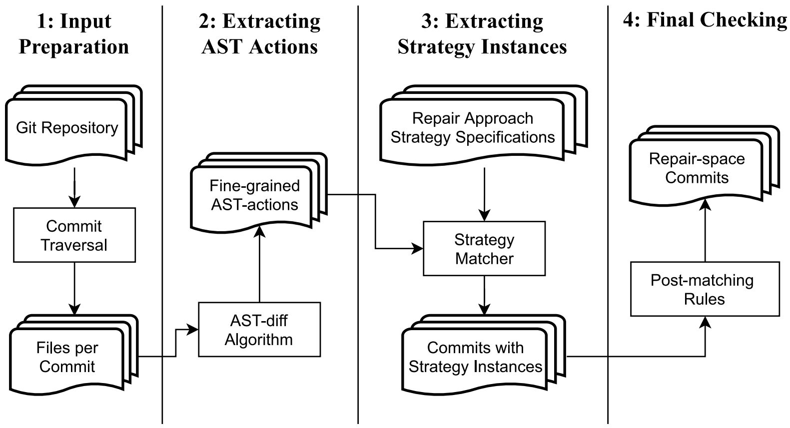

Fig. 1. Overview of the approach.

Repair-space Commit: Given a commit c that transforms the *old_version* of a program into its *new_version* and a repair approach r, we say that commit $c = (old\_version, new\_version)$ is a *repair-space commit* for r if and only if *new_version* is in the search space of r when *old_version* is given as the input.

Consider Listing 1, which is a real commit in project Jgrapht[1]. This commit is an example of repair-space commits of the repair search space of NPEfix (Cornu et al., 2015). The commit contains a typical NPEfix null check characteristic of its search space. Hence we say that this commit is a repair-space commit for NPEfix. In this example, line 1 and line 5 are from the old_version, while in the new_version lines 2–4 are added. According to the definition of NPEFix, new_version is in its search space because NPEfix has a "skip method" strategy that is able to produce this patch.

Commit Coverage (CC): The *commit coverage* of repair approach r over a set of commits S is the number of commits in S that are repair-space commits for r divided by the total number of commits in S.

We consider real human-written patches to be useful patches. Therefore, if a repair approach has a high commit coverage over a large dataset of human-written patches, this indicates the potential usefulness of the actual implementation of the repair approach.

In this section, we design a framework to detect repair-space commits. When the repair-space commits are detected, computing the commit coverages is trivial. In this framework, we encode each repair strategy by specifying (a) the repair operators from the strategy, expressed using fine-grained code changes, and (b) the rules that those changes must respect (e.g., the code introduced by a change is a valid ingredient according to a given scope).

Fig. 1 shows an overview of how the proposed approach works. The whole process consists of four steps. (1) Input preparation: for a given Git repository, we identify the updated files for each commit (see Section 4.3). (2) Extracting AST actions: for each updated file, the actions that transform the AST of the old version into the new one are extracted (see Section 4.4). (3) Extracting strategy instances: the updated files whose corresponding AST actions match a "strategy specification" (which is defined in Section 4.5.1) are determined (see Section 4.5). We design the strategy specifications to model the repair strategies employed by the considered repair approaches. (4) Final check: the commits whose updated files match strategy specifications are checked for additional constraints (see Section 4.6). The result of this last step are the detected repair-space commits.

```
1   if (var == val1){
2     var = val1 + val2;
3   } else if (var == val2) {
4     var = val1;
5   } else {
6 -    var = val1 + val2;
7 +    var = val1;
8   }
```

Listing 2: A code change which could be seen as a "statement replacement", "expression replacement", or "operator removal".

4.2. Challenges

The major challenge of repair-space commit detection is to find a representation of AST modifications that is appropriate for capturing program repair strategies. For example, consider Listing 2. The AST modifications in this example can be represented in many different ways incl.: (1) R1: It can be seen as a "statement replacement" action: a statement (line 6) is replaced by a new statement (line 7). The new statement is copied from line 4 of the same file. (2) R2: It can be seen as an "expression replacement" action: an expression ("val1+val2") from line 6 is replaced by a new expression ("val1"). The new expression is copied from line 4 of the same file. (3) R3: It can be seen as an "operator removal" action: an operator and the corresponding operand ("+ val2") is removed from line 6 and the statement at line 7 is the result.

If the code change at Listing 2 is represented as R1 (i.e., replace the statement in line 6 by another one), it lies in the search space of Arja and GenProg because those approaches are able to generate patches that replace one buggy statement by another. If it is represented with R2 (i.e., replacement of an expression by another one), it lies in the search space of Cardumen because Cardumen repairs bugs by replacing expressions. Finally, if it is represented with the third option, it does not lie in the search space of any of these three repair approaches because none of them has a repair operator that consists in removing a '+' operator. To sum up, one of the main research challenges that we are addressing in this study is to find the right AST action representation, which is appropriate for specifying repair search spaces.

[1] https://github.com/jgrapht/jgrapht/commit/275c6fdb.

Algorithm 1 Algorithm of the proposed approach.

```
Inputs:
    git_repo: The given Git repository
    repair_approach: The repair approach whose
    search space should be considered
Outputs:
    repair_space_commits: The set of detected
    repair-space commits for repair_approach
1:  commits ← get_commits(git_repo)
2:  specs ← get_specifications(repair_approach)
3:  for each commit c in commits do
4:      SI ← [] /* SI: strategy_instances */
5:      if only_one_file_updated(c) then
6:          f ← get_updated_file(c)
7:          f_p ← get_previous_version(f)
8:          f_n ← get_new_version(f)
9:          ES ← GetDiff(f_p, f_n) /* ES: Edit Script */
10:         for each specification s in specs do
11:             if match(ES, s) then
12:                 SI.insert(ES)
13:     if pass_post_rules(SI, c, repair_approach) then
14:         repair_space_commits.insert(c)
```

4.3. Input preparation

Algorithm 1 shows how our technique works. It takes as input the path to a Git repository and a repair approach whose search space should be considered. Then, it traverses over the history of the repository from the oldest commit to the most recent one.

For each commit c, LighteR checks that only one file is updated (line 5). Changes in multiple updated files cannot be covered by a single strategy instance, while we only target single-instance fixes that lie in the repair search space (see Section 4.5.2). Therefore, we discard commits with multiple updated files. If c has one updated file, LighteR gets that file f (line 6) and constructs a pair of files $<f_p, f_n>$, where $<f_p>$ is the version of f previous to c (retrieved in line 7), and $<f_n>$ is the new version obtained after c (retrieved in line 8).

4.4. Extracting AST actions from updated files

In the second step, LighteR computes the AST differences between the pair of files $<f_p, f_n>$ (line 9). The output of this step is an *edit script (ES)*, a list of actions that transforms f_p into f_n.

LighteR uses GumTree algorithm (Falleri et al., 2014) to compute these actions at the AST level. In GumTree, there are four types of action: (1) update, which changes the value of an AST node, (2) insert, which inserts a new AST node, (3) delete, which deletes an existing AST node, and (4) move, which moves an AST node and makes it child of another node. These extracted fine-grained AST actions are passed to the next step.

4.5. Extracting strategy instances

This step determines if the fine-grained AST actions from an edit script correspond to those that can be synthesized by a repair approach.

For each repair approach, we come up with one or more strategy specifications (described in Section 4.5.1) that define its search space. Strategy specifications are abstract representations of the repair strategies employed by repair approaches.

If AST actions in an edit script ES match with a strategy specification s, we say that the ES is an *instance* of the s.

We now describe the specification language and then the matching process.

```xml
1 <pattern name="binary_upd">
2  <entity id="1" type="If"/>
3  <entity id="2" type="BinaryOperator">
4   <parent parentId="1">
5  </entity>
6  <action entityId="2" type="UPD"/>
7 </pattern>
```

Listing 3: One of the strategy specifications for jMutRepair.

4.5.1. Strategy specification

Each specification uses an abstract representation to specify a certain repair strategy of a program repair approach. The specifications are represented in the *change pattern specification language* of Martinez and Monperrus (2019), which we now briefly present. A strategy specification consists of a set of *actions*, and each action is performed on an *entity*. The types of actions of specifications are the same as the types of AST actions that GumTree extracts (*update*, *insert*, *delete*, and *move*). In addition to these action types, a strategy specification can also contain an action of type *unchanged*, which indicates an entity should not be changed (i.e., not affected by any action). Finally, a strategy specification can also define parenthood relations between entities.

For example, Listing 3 is a specification that corresponds to a repair strategy used by jMutRepair (Martinez and Monperrus, 2016). According to this strategy, a binary operator inside an if-condition can be changed to another operator. Line 6 of Listing 3 represents the update action. As it is stated, the "entityId" of the subject entity is "2". Therefore, this action is performed on the entity defined in line 3. As shown in the specification, the type of this entity is "BinaryOperator" and the id of its parent is "1" (see line 4). Finally, the parent entity is defined in line 2 and as it is mentioned there its type is "If".

4.5.2. Strategy specification matching

For each strategy specification s of a repair approach, LighteR checks if s matches with the AST actions (ES) previously computed (line 11 of Algorithm 1). To this end, for each action A_p specified in s, we check whether there exists an actual action in ES that affects the nodes specified by A_p. The details of the matching process can be found in the study of Martinez and Monperrus (2019).

Note that LighteR considers a commit to be in the search space of repair approach r, only if all the changes in the commit are covered by a single strategy instance of r. We call such commits, *single-instance* fixes. For example, a fix by GenProg that removes multiple statements from different methods is not considered as a repair-space commit by LighteR. On the other hand, a fix that removes a single statement that contains multiple-lines is indeed considered to be in the search space of GenProg by LighteR. It is worth mentioning that LighteR already has the potential to detect multi-instance fixes as well. However, possible overlaps between multiple strategy instances in a single fix can lead to a high level of noise in LighteR detection algorithm. Since most existing repair tools create single-instance fixes in practice (Durieux et al., 2019), we ignore multi-instance commits in the current version of LighteR.

4.6. Final checking

In order to make sure that the source code changes from the identified commits lie in the search space of detected repair approaches, we also check particular rules that repair approaches

```
<pattern name="genprog_simple">
  <entity id="1" type="*" role="Statement"/>
  <action entityId="1" type="*"/>
</pattern>
```

Listing 4: Simple specification for GenProg strategies.

```
<pattern name="genprog_indirect">
  <entity id="1" type="*" role="Statement"/>
  <entity id="2" type="*" role="*">
    <parent parentId="1" distance="100000"/>
  </entity>
  <action entityId="2" type="*"/>
</pattern>
```

Listing 5: Second specification for GenProg strategies.

follow for generating patches (line 13 of Algorithm 1). We call these rules the *post-matching rules*. These rules determine how a repair approach synthesizes new code.

The post-matching rules can be divided into two groups: (1) rules specifying how the ingredients are extracted from the considered scope, and (2) rules specifying how the ingredients are merged together to form new code fragments that are used in the patch.

As an example, Cardumen (Martinez and Monperrus, 2018) considers all the variables and literals in the scope as repair ingredients. Next, it takes an existing *expression* and replaces its variables and literals with extracted ingredients of the same type to make a new expression. This new expression is then used to generate a patch.

The commits given as the input of this step that follow the post-matching rules are considered as the detected repair-space commits.

4.7. Meeting the challenge

As explained in Section 4.2, the modifications of an AST in a commit can be represented in various ways. The edit script that GumTree produces is only one of such representations. To make sure that we recognize any possible correspondence between an edit script and a repair strategy, we take two steps as follows.

(1) Designing all relevant specifications: We design all possible specifications whose matching edit scripts can be an instance of the target repair strategy. The "`get_specifications`" method at line 2 of Algorithm 1 retrieves all of these specifications. For example, consider the repair strategies of GenProg. 4 is the simple and natural specification that encodes those strategies. It represents any action (move/insertion/removal) on a statement. The code change in Listing 2 is inserting a new statement (at line 7), therefore it is an instance of this simple specification for GenProg strategies. However, the edit script *ES* that GumTree generates for this change is an "operand removal" action that removes "`+val2`". *ES* does not match the simple specification, as the specification requires an action on a statement, while *ES* contains an action on an operand.

To make sure that LIGHTER does not miss strategy instances such as Listing 2, we also design and add the second specification shown in 5 for GenProg. This specification represents an action (move/insertion/removal) on an element inside a statement. This specification matches with the *ES*, since removing "`+val2`" from line 6 of Listing 2 is counted as a removal of an operand inside the statement "`var=val1+val2;`". Therefore, by adding the specification in Listing 2, we also catch this strategy instance.

(2) Filtering out non-instance matches: By designing and considering all relevant specifications, we may match *ES* with a specification for a repair strategy *rs*, while the code change represented by *ES* is not an instance of *rs*. In our final checking step, we also check whether each matched edit scripts represents a change that is actually a strategy instance. This final check should be particularly tailored for each repair strategy. For example, for GenProg, we make sure the new statement is similar to an existing statement before repair.

Specifications designed in the first step match only a few types of commits and filter out the rest. This enables us to carefully design proper checks for the commits that go to the "filtering out non-instance matches" step. This whole process is a fast and accurate mechanism for matching edit scripts and specifications.

4.8. Repair approaches considered

In LIGHTER, we specify the search space of test-based repair approaches according to the following criteria. First, they are included in Table 1 of Durieux et al. (2019) study. Durieux et al.'s study provides a large list of repair tools that is frequently used by researchers as a reference for conducting empirical analysis of program repair (Qin et al., 2021; Aleti and Martinez, 2021; Lin et al., 2020). Second, the article presenting the repair approach provides us with enough information to specify the repair search space. Third, the repair approach should have an explicitly defined search space so that we can specify it. For example, we exclude learning-based approaches, as their search space is hidden in the weights of their neural network. According to those criteria, we select eight repair approaches: Arja (Yuan and Banzhaf, 2018), Cardumen (Martinez and Monperrus, 2018), Elixir (Saha et al., 2017), GenProg (Le Goues et al., 2012a), jMutRepair (Martinez and Monperrus, 2016), Kali (Qi et al., 2015), Nopol (Xuan et al., 2016), and NPEfix (Cornu et al., 2015). LIGHTER tries to detect Java repair-space commits (the dataset is introduced in Section 5.2). Therefore, we consider the implementations of these approaches that repair Java programs. This means for GenProg and Kali, which have implementations for both Java and C, we consider their implementations for Java in jGenProg2 and jKali (Martinez and Monperrus, 2016).

In Table 2, each row presents a brief overview of the strategy specifications and the post-matching rules that we consider to encode the search space of the corresponding repair approach. For instance, three strategy specifications are considered to encode the repair strategies employed by Arja, one for inserting a new statement, one for removing a statement, and one for replacing a statement. Moreover, in accordance with the process of synthesizing new statements in Arja, we have a post-matching rule: the new statement should be a copy of an existing statement, while the variables, literals, and methods can be replaced by other variables, literals, and methods in the scope.

4.9. Implementation

We implement a prototype of our proposed method called LIGHTER. LIGHTER is built on top of Coming (Martinez and Monperrus, 2019). Coming is designed to mine instances of code change patterns in Git repositories. LIGHTER extends Coming by adding strategy specifications and post-matching rules for the considered repair approaches. The post-matching rules are implemented in Java and the strategy specifications are represented in the *change pattern specification language* (Martinez and Monperrus, 2019) as noted in Section 4.5.1. For all approaches based

Table 2
Specification of the Search Space of 8 Notable Repair Approaches.

Name	Excerpt of strategy specifications	Excerpt of post-matching rules
Arja	Removing a statement; Inserting a new statement; Replacing a statement with a new one.	The new statement should be a copy of an existing statement, while the variables, methods, and literals can be replaced by other variables, methods, and literals of the same type in the scope.
Cardumen	Replacing an expression with a new expression.	The new expression should be a copy of an existing expression, while the variables, and literals can be replaced by other variables, and literals of the same type in the scope. Moreover, the new expression should have the same return type as the old one.
Elixir	Replacing the declaration type of a variable with a wider one (e.g float to double); Replacing a return expression with a new expression; Moving a statement into a new if-statement. The condition of the new if-statement checks if one of the variables used in the statement is not null; Mutating a binary operator. e.g., "<" to ">"; Replacing a method invocation with a new one; Inserting a new method invocation; Removing a predicate from the boolean expression of an if-condition or return statement; Adding a predicate to the boolean expression of an if-condition or return statement; Moving a statement into a new if-statement. The condition of the new if-statement checks if an array or collection access is in a range.	All the code fragments used in a synthesized code should be collected from existing code. Specifically, a new method invocation should call a method that is already called in the scope. The argument list of a new method invocation should also be a list of literals and variables in the scope. For more details, please see the original paper by Saha et al. (2017).
GenProg	Removing a statement; Inserting a new statement; Replacing a statement with a new one.	The new statement should be exactly a copy of an exiting statement in the scope.
jMutRepair	Changing a unary or binary operator inside an if-condition.	No post-matching rule.
Kali	Removing a statement; Changing an if-condition to *true/false*; Inserting a return statement.	No post-matching rule.
Nopol	Replacing an if-condition with a new if-condition; Inserting a new if-statement and moving an existing statement into it.	The new if-condition should consist of variables, methods, and literals that exist in the scope.
NPEfix	Moving a statement into a new if-statement. The if-condition checks if a variable used in the moved statement is not null; Moving a statement into a new if-statement. The if-condition checks if a variable used in the moved statement is not null. Also, add an else-block which returns "null" or a new object or another variable of the desired type; Moving a statement into a new if-statement. The if-condition checks if a variable used in the moved statement is not null. Also, add an else-block which executes the same statement but replaces the checked variable with a new object or another variable of the same type; Inserting a new if-statement before a target statement. The if-condition checks if a variable used in the target statement is null. The corresponding then-statement sets the value of the checked variable to a new object or another variable of the same type.	The variables used in synthesized code should be from existing code in the scope.

on code reuse, LIGHTER considers that the scope of ingredients is the *same file* level. This means a repair-space commit must utilize ingredients from the same file as the repair location. Note that previous studies show that among human made patches that reuse existing ingredients, in 65% of the cases all ingredients are selected from the exact same file (Yang et al., 2021). Therefore, we select the file scope as it helps us simplify the experiments without loss of generality. For sake of open science, LIGHTER is made publicly available (Etemadi et al., 2021).

5. Experimental methodology

5.1. Research questions

In this paper, we study five research questions. The first two concern a deep study of the commit coverage of program repair approaches over human-written past commits.

- RQ1: How do repair approaches compare to each other in terms of human-written patches that lie in their search spaces?
  To answer this research question, we use LIGHTER to detect repair-space commits for the considered repair approaches, and thereby compute their commit coverage over a large dataset of real-world commits. Moreover, we conduct a manual study to measure the prevalence of repair-space commits that are actually fixing bugs.
- RQ2: To what extent do the search spaces of repair approaches overlap according to our lightweight analysis?
  It is known that some repair strategies are shared between repair approaches (Martinez and Monperrus, 2019a). Therefore, one can expect to see shared commits between search spaces. In this experiment, we investigate the extent of this search space overlap.

The next two research questions measure the accuracy of the matching mechanisms and the search space specifications employed by LIGHTER.

- RQ3: What is the recall of LIGHTER for repair-space commit detection?

Table 3
Dataset features.

PB		#Commits
All 72 repos		55,309
Repos with at least 1000 commits		43,000
Repos with fewer than 1000 commits		12,309
Ex: github.com/apache/pinot		1,000
Ex: github.com/2018swecapstone/h2ms		931
Ground-truth	**#Bugs**	**#Patches**
All 5 projects	160	729
jfreechart	15	112
closure-compiler	54	91
commons-lang	25	165
commons-math	56	319
joda-time	10	42

- RQ4: What is the precision of LighteR for repair-space commit detection?

The last research question studies the complexity of search space specification in LighteR.

- RQ5: How complex are the commit matching criteria that encode the repair search space of program repair approaches? LighteR specifies the search space of eight existing repair approaches. To study the difficulty of encoding other repair approaches, we analyze the complexity of strategy specification and post-matching rules already implemented.

5.2. Datasets

In this paper we use two datasets: (1) a curated set of open-source repositories and their commits. This dataset is used to answer RQ1, RQ2, and RQ4. (2) a set of patches (i.e., source code changes) that are generated by automatic program repair approaches. We employ this dataset to answer RQ3. They are collected as follows.

The dataset of repositories, which we call PB, contains all projects that are included in the bug benchmark Bears (Madeiral et al., 2019). Bears contains bugs and their respective patches collected from 72 distinct open-source Java projects. We consider Bears since it has the largest number of projects among datasets of its type (72 versus 6 for Defects4J (Just et al., 2014)).

For each project in PB, we consider the last 1000 commits as of October 08, 2021. For projects with less than 1000 commits, we consider all commits. As shown in Table 3, the total number of commits considered in this dataset is 55,309. 43,000 commits are selected from 43 projects with at least 1000 commits and the remaining 12,309 commits are selected from projects with fewer than 1000 commits. For example, "pinot" has 1000 commits in PB, while "h2ms" has no more than 931 commits.

It is to be noted that PB contains all types of commits. This is a deliberate decision because there is no accurate automatic oracle for determining bug-fix commits. Also, manual detection of bug-fix commits in a very large dataset of commits is practically impossible.

The second dataset, named Ground-truth, is a benchmark of patches produced by the eight automatic repair approaches that we consider in this work. This dataset is built as a subset of the dataset DRR (Ye et al., 2021a) and from the dataset NPEfix (Durieux et al., 2017). We choose DRR because it is the largest curated collection of patches automatically generated by repair approaches on the Defects4J dataset (Just et al., 2014). It includes patches for all approaches we consider, except for NPEfix. Note that DRR also includes patches generated by approaches that we do not consider, those patches are not included in Ground-truth. Second, the NPEfix patches come from the original study of Durieux et al. (2017), we use them because DRR does not contain patches for NPEfix while bug-fix commits related to null-pointers are important in practice.

In total, Ground-truth contains 729 patches. As presented in Table 3, these patches are generated for 160 different bugs. The number of bugs and patches per project are also shown in Table 3. All five projects and their bugs are from the well-known Defects4J (Just et al., 2014) dataset and the patches come from the carefully curated DRR dataset (Ye et al., 2021a) and NPEfix study (Durieux et al., 2017). For example, 15 bugs from "jfreechart" and their 112 fixing patches are considered.

5.3. Protocol for RQ1

RQ1: How do repair approaches compare to each other in terms of human-written patches that lie in their search spaces?

We use LighteR to answer RQ1. We run LighteR on all commits collected in the PB dataset. This experiment is carried out in the form of a sequence of executions. Each execution is represented by a pair, like <*a*, *r*>, where *a* is a repair approach and *r* is a repository. In total, we perform 576 (8 approaches $\times$ 72 projects) executions. In each execution, LighteR goes through the commits of *r* and checks if the source code changes in each commit are in the search space of *a*. This experiment is conducted on a server with an Intel Core Processor running at 2299.996 MHz using 8 GB of RAM, running Ubuntu version 18.04.

The result of this experiment is a set of repair-space commits for each pair of repair approach and repository. Based on those results, we compute the commit coverage for each repair approach to find which of them perform repairs that are similar to human-made changes in open source projects. In addition to calculating the percentage of all commits that are covered by the considered repair approaches, we compute the ratio of commits with changes in exactly one source file that lie in the search space of repair approaches. This gives use a better sense of the coverage of the considered repair approaches over the particular commits that we target in this experiment.

Next, we perform a comprehensive manual analysis to identify the detected repair-space commits that are bug-fixing. The process is as follows: (1) each detected repair-space commit is labeled by two persons, (2) the labels could be "bug-fixing", "not-bug-fixing", or "dont-know"; (3) in case of a judgment conflict between the two persons, a third person annotates the commit to break the tie. Ten people participated in this annotation process, seven Ph.D. students, one Postdoc researcher, and two professors, all working on areas close to automatic program repair. The results of this manual analysis tell us how many of the repair-space commits detected by LighteR are bug fixes. The manual analysis process is extensively documented in our companion repository (Etemadi et al., 2021).

5.4. Protocol for RQ2

RQ2: To what extent do the search spaces of repair approaches overlap according to our lightweight analysis?

To answer RQ2, for each repair approach, we compute the number of its repair-space commits that also lie in the search space of another repair approach. For this purpose, we consider the repair-space commits that are detected in response to RQ1.

Moreover, we group the repair approaches by the type of changes they apply and analyze the overlap between the search space of these groups. This analysis reveals well-known types of changes that repair approaches apply and cannot be replaced by other types of changes. By considering the repair strategies listed in Table 2 and per authors' consensus, we categorize repair approaches into three groups. Three of the repair approaches make a change in if-conditions of the subject program (jMutRepair, Nopol, and NPEfix) and three of them add/remove/replace statements (Arja, GenProg, and Kali). We call these groups "if-change" and "s-change" (for statement change), respectively. We put the remaining two approaches (Cardumen and Elixir) in a third group and call that group "other". Note that Kali removes a functionality in program, which can be done either by removing a statement, inserting a return statement, or changing an if-condition to `true` or `false`. This means Kali is able to make a change on both statements and if-conditions. However, by manually reviewing our early results, almost all Kali repair-space commits remove a statement. Thus, we put it in the s-change group. Finally, we compute the overlap of repair-space commits among these three groups.

If the results of this experiment show a repair approach has many unique commits, it means that its adds something useful and original capabilities to the state-of-the-art of automatic program repair.

5.5. Protocol for RQ3

RQ3: What is the recall of LighteR for repair-space commit detection?

To answer RQ3, we design an experiment to determine the recall of LighteR, that is, how many of the patches actually generated by a repair approach are correctly detected by LighteR. For this, we run LighteR on the Ground-truth dataset because we need a fully labeled dataset. For the same reason, it is impossible to use PB in RQ3 because the commits in PB are not labeled.

Ideally, LighteR should be able to detect all of these source code changes as instances of corresponding repair approaches. However, it might happen that some ground-truth patches cannot be detected by LighteR due to the difficulty of encoding the search space of repair approaches. We call these patches the *false negative patches (FN)*.

Finally, the recall of LighteR is calculated according to Eq. (1). In this equation *GT* represents the set of all patches in the Ground-truth. *TP* is the set of *true positives*: the set of ground-truth patches that are detected as repair-space commits by LighteR. Note that $|TP| = |GT| - |FN|$ because *TP* is the set of ground-truth patches that are not among the false negatives.

$$recall = \frac{|TP|}{|GT|} \tag{1}$$

Since LighteR is the first tool of its type, there is no other work that we can directly compare against. However, a close tool is PPD (Patch Pattern Detector), which detects instances of repair patterns (Madeiral et al., 2018). The repair patterns that PPD looks for are extracted from the code changes in Defects4J dataset and the tool is also evaluated on Defects4J. We compare the recall of our tool against PPD.

5.6. Protocol for RQ4

RQ4: What is the precision of LighteR for repair-space commit detection?

We measure the precision of LighteR as follows: we randomly select a sample of *n* repair-space commits for each tool in the PB dataset. Next, we carry out a manual analysis to decide if the detected repair-space commits actually lie in the search space of corresponding repair approaches or not.

We select a value of *n* such that the overall manual work stay under two days over all analysts. Recall that the annotators have to be trained to be fully familiar with the corresponding repair approaches. According to this criterion, we select 30 commits per repair approach, each of them being analyzed by three analysts.

This manual analysis is made by seven analysts in total, all of whom are researchers in the field of automatic program repair: three Ph.D. students, two postdoctoral researchers, and two professors. Each commit is annotated by two analysts. If the first two annotations conflict with each other, a third analyst annotates to break the tie. All results from this experiment are publicly available (see Etemadi et al., 2021).

The precision for each repair approach is computed per Eq. (2). In this equation, *true positive (TP)* represents the set of detected repair-space commits that are actual repair-space commits according to the manual investigation. Moreover, *RSCommits* is the set of all considered repair-space commits for the current repair approach.

$$precision = \frac{|TP|}{|RSCommits|} \tag{2}$$

Similar to the recall, we also compare the precision of our tool with that of PPD (Madeiral et al., 2018).

5.7. Protocol for RQ5

RQ5: How complex are the commit matching criteria that encode the repair search space of program repair approaches?
To answer RQ5, we compute relevant metrics for strategy specifications and post-matching rules (see Sections 4.6 and 4.5.1). First, for measuring the complexity of a repair approach encoding, we define 3 variables: (1) total number of specifications, (2) number of actions and (3) number of entities inside the specifications. When a repair approach has more than one corresponding specifications, the number of actions/entities for that approach is the sum of the number of actions/entities in all the related specifications. We consider the lines of code (LOC) to measure the complexity of post-matching rules.

6. Experimental results

We now present our experimental results on the commit coverage of program repair approaches.

6.1. RQ1: Commit coverage per repair approach

Table 4 shows the results of our first experiment. In this table, each row represents the data for one repair approach. The "#RSC" column shows the number all commits that LighteR detects as repair-space commits. For each approach, "%CC" presents the commit coverage, which is equal to the percentage of all 55,309 human-written commits that are considered to be repair-space commits for that approach. "%TCC" is the percentage of our targeted commits that (a) have changes in exactly one source file, and (b) lie in the search space of a repair approach. "BF", "NBF", and "DN" indicate the number and percentage of repair-space commits labeled as "bug-fix", "not-bug-fix", and "dont-know", respectively, per our manual analysis described in Section 5.3. Finally, the "Exec. time" column represents how many seconds it takes on average for LighteR to check if a commit is in the search space of the corresponding approach. For example, 263 of commits are detected to be in the search space of Arja, which means Arja covers 0.47% of all commits and 3.46% of commits with changes in exactly one source file. Moreover, 51% of Arja's repair-space commits are labeled as "bug-fix" commits by the annotators.

In total, $747/55,309\,(1.35\%)$ commits are detected as repair-space commits for, at least, one of the repair approaches. Among the considered repair approaches, the top two approaches in

```
- RouteHandler rtHandler = route.getRouteHandler();
  route.getRouteHandler().handle(req, res, this);
```

Listing 6: Example of not-bug-fix repair-space commits.

Table 4
RQ1: The presence of repair-space commits in 72 open-source projects.

Approach	#RSC[a]	%CC[c]	%TCC[c]	BF[d]	NBF[d]	DN[d]	Exec. Time
Arja	263	0.47	3.46	51%	38%	11%	0.76 s
Cardumen	219	0.39	2.88	63%	29%	8%	0.33 s
Elixir	369	**0.66**	4.86	67%	24%	9%	1.72 s
GenProg	181	0.32	2.38	46%	42%	12%	0.77 s
jMutRepair	7	0.01	0.09	86%	14%	0%	0.87 s
Kali	117	0.21	1.54	31%	56%	13%	0.29 s
Nopol	174	0.31	2.29	81%	13%	6%	1.34 s
NPEfix	33	0.05	0.43	90%	3%	7%	0.46 s
All	747[b]	1.35	9.85	62%	29%	9%	0.81

[a]RSC stands for "repair-space commits". This column shows how many of the 55,309 commits that are analyzed against the search space of all tools are detected as repair-space commits of this approach.
[b]747 commits are detected repair-space commits for at least one repair approach. Note that this is not the sum of numbers in this column.
[c]%CC is the commit coverage of the corresponding approach. Also, %TCC shows the percentage of commits with changes in exactly one source file that lie in the search space of corresponding repair approach. The total number of these commits is 7,583. For example, Elixir covers 0.66% (369/55,309) of all considered commits and 4.86% (369/7,583) of commits with changes in exactly one source file.
[d]BF, NBF, and DN represent the number of repair-space commits labeled as "bug-fix", "not-bug-fix", and "dont-know", respectively.

terms of the commit coverage are Arja and Elixir. Given the strategies used by these approaches, this confirms the results of Martinez et al. (2014) showing that in a significant number of commits all the new lines are copied from the previous versions of the same file.

The majority (62%) of the detected repair-space commits are labeled as "bug-fix". This is in line with the fact that the encoded repair strategies are indeed related to the activity of bug fixing.

Interestingly, there is also a notable portion of the repair-space commits (29%) that are not considered as bug-fixing, yet can likely be generated by a repair approach. This suggests that the considered repair approaches can also be used for purposes other than bug-fixing. We manually analyze them and identify that there are two common types of "not-bug-fix" repair-space commits: (1) commits that only change logging outputs, and (2) commits that remove unused code. For example, 6 is a commit from the "pippo" project that removes an unused variable "rtHandler". Although this commit is not labeled as bug-fix, it could be produced by Arja, GenProg, and Kali.

There are also 9% of repair-space commits that the analysts could not determine if they were bug-fixing or not. These repair-space commits are labeled as "dont-know". Most of these commits are hard to label due to uninformative commit messages and changes in uncommented parts of the code. This confirms that assessing arbitrary commits from the field is an inherently difficult task (Martinez and Monperrus, 2015; Le et al., 2019). The number of unclassifiable commits is low, much lower than the classified ones (9% versus 91%).

Now, let us discuss the execution speed of our lightweight analysis. The average time spent to check if a commit is in the search space of a repair approach is 0.81 s. Elixir has the slowest strategy specifications and post-matching rules, LIGHTER needs 1.72 s on average to analyze if a commit is in the search space of Elixir. On the contrary, the LIGHTER configured for Kali takes only 0.29 s per commit. Those numbers indicate that LIGHTER can scale to large repositories: for instance, analyzing 3.3k commits (the maximum number of commits for 99% of Java projects on Github) against the search space of all tools would take approximately 6 hours, which is acceptable given that it is one-short computation task.

Answer to RQ1: How do repair approaches compare to each other in terms of human-written patches that lie in their search spaces?
According to our analysis, 1.35% (747/55,309) of commits from 72 projects of dataset PB are detected as being in the search space of at least one repair approach. This experiment shows that our novel method enables researchers and practitioners to evaluate the potential of repair approaches in terms of their commit coverage. To the best of our knowledge, we are the very first to use commit coverage to compare program repair strategies.

6.2. RQ2: Overlap between repair approaches

Table 5 shows the proportion of overlapping repair-space commits between each pair of repair approaches. Each cell presents the percentage of detected repair-space commits in the search space of an approach that also lie in the search space of another approach. For instance, 64% of the repair-space commits of Arja (row 1) also lie in the search space of GenProg (column 2). On the opposite side, 93% of repair-space commits of GenProg (row 2) also lie in the search space of Arja (column 1). In the cells on the diagonal, the cell content represents the number and percentage of unique repair-space commits of the corresponding approach. A unique repair-space commits is a commit that only lies in the search space of one single repair approach and is not covered by other approaches.

The results from Table 5 show that the ratio of shared repair-space commits vary significantly among repair approaches ranging from 0% to 94%. The table also shows that there are repair approaches with significant ratio of unique repair-space commits, clearly higher than others. For example, NPEfix Cardumen, and Elixir have more than 40% unique repair-space commits, while this number is less than 10% for GenProg and Kali. In total, 47% (357/747) of the repair-space commits are unique and the remaining 53% (390/747) lie in the overlapping parts of the search spaces. Here, uniqueness is a proxy to value, the approaches with high percentage of unique repair-space commits cannot be replaced by any other approaches.

As explained in Section 5.4, we also divide the approaches into three groups (*if-change*, *s-change*, and *other*) and compare their search spaces against each other. Fig. 2 depicts the overlaps between the search space of these groups of repair approaches. Note that the data represented in Fig. 2 cannot be fully retrieved from Table 5. Recall that the search space of each group is the union of the search spaces of its members. The numbers on this figure show how many of the detected repair-space commits lie in the corresponding group. For example, 170 are exclusively in the search space of *s-change* group and 112 are common between repair-space commits of *if-change* and *other* but do not lie in the search space of *s-change*. The overlap between if-change and s-change is small. There are 10 (=8+2) commits shared between if-change and s-change, 8 of them are also shared with "other", while 2 of them are outside the search space of other. The total number of all repair-space commits in if-change and s-change is 459 (=67+2+8+170+100+112), therefore only 2% (10/459) of their commits are common. The small overlap between if-change and s-change indicates that those two groups are complementary.

Moreover, as illustrated in Table 5, the overlap between approaches of a group is usually higher than the overlap between

Table 5
The overlapping of the repair approaches. Each row presents the percentage of detected repair-space commits in the search space of the corresponding approach that also lie in the search space of the rest of the approaches. For instance, 64% of the repair-space commits of Arja (row 1) also lie in the search space of GenProg (column 2). Numbers on the table diagonal represent the unique repair-space commits, which means the commits that only lie in the search space of the corresponding repair approach. For example, 12% of the commits in the search space of Arja are unique, they do not lie in the search space of any other repair approach. The color indicates overlap, the darker the cell the more the overlap.

	S-change Arja	GenProg	Kali	If-change jMutRepair	Nopol	NPEfix	Other Cardumen	Elixir
Arja	**12% (33)**	64% (170)	42% (111)	0% (0)	3% (8)	0% (1)	20% (54)	32% (85)
GenProg	93% (170)	**2% (4)**	60% (110)	0% (0)	3% (6)	0% (0)	14% (27)	20% (38)
Kali	94% (111)	94% (110)	**5% (6)**	0% (0)	0% (0)	0% (0)	0% (0)	0% (1)
jMutRepair	0% (0)	0% (0)	0% (0)	**14% (1)**	85% (6)	0% (0)	0% (0)	85% (6)
Nopol	4% (8)	3% (6)	0% (0)	3% (6)	**29% (52)**	10% (19)	17% (30)	55% (96)
NPEfix	3% (1)	0% (0)	0% (0)	0% (0)	57% (19)	**42% (14)**	0% (0)	57% (19)
Cardumen	24% (54)	12% (27)	0% (0)	0% (0)	13% (30)	0% (0)	**42% (94)**	36% (80)
Elixir	23% (85)	10% (38)	0% (1)	1% (6)	26% (96)	5% (19)	21% (80)	**41% (153)**

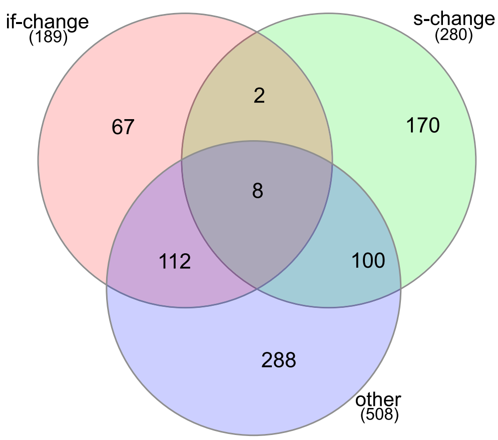

Fig. 2. Overlaps between the search space of groups of repair approaches.

Table 6
RQ3: Recall for each repair search space.

Approach	#GT	#TP	Recall
Arja	129	117	0.90
Cardumen	129	118	0.91
Elixir	37	31	0.83
GenProg	116	101	0.87
jMutRepair	52	52	1
Kali	53	47	0.88
Nopol	103	101	0.98
NPEfix	110	107	0.97
Total	729	674	0.92

```
1  if (this.autoSort) {
2 -  this.data.add(-index-1, new XYDataItem(x,y));
3 +  this.data.add(new XYDataItem(x,y));
4  }
```

Listing 7: Example of an undetected ground-truth patch.

approaches from different groups. For example, Kali has 111 (94%) repair-space commits in common with other approaches from the s-change group, while it has no repair-space commit in common with approaches from if-change group. These findings confirm that the grouping of approaches is meaningful.

Implication: Based on the results from this experiment, we conclude that a general purpose repair approach should have at least three types of repair strategies: (1) statement replacement (s-change), (2) control-flow mutator (if-condition), and (3) expression replacement (Cardumen and Elixir). That is because no strict subset of these three categories covers the overall search space.

Answer to RQ2: To what extent do the search spaces of repair approaches overlap according to our lightweight analysis?
Our results show that 47% (357/747) of repair-space commits are unique to one specific repair approach, which is a novel result in the literature. NPEfix, Cardumen, and Elixir have the largest number of unique repair-space commits (more than 40%). This little overlap shows that program repair research is producing approaches that are complementary in practice, and useful for practitioners in an integrated manner.

6.3. RQ3: Recall of LIGHTER

Table 6 studies the recall of LIGHTER. Columns "#GT" and "#TP" indicate the number of ground-truth and true positive patches, respectively (see Section 5.5 for more details). The recall is computed according to Eq. (1).

For instance, there are 129 ground-truth patches for Arja and LIGHTER detects 117 of them. Consequently, the recall for Arja is 0.90. LIGHTER has the lowest recall for Elixir (0.83) and the highest one for jMutRepair, a perfect recall of 1. The total recall is 0.92, this is arguably a high recall, in line with the state-of-the-art related tool PPD (Madeiral et al., 2018) (the total recall of PPD is 0.92 as well).

Listing 1 is an example of a NPEfix ground-truth patch that is detected by LIGHTER. In contrast, 7 shows an example of a GenProg ground-truth patch that is not detected. In this example, GenProg replaces the statement in line 2 with the statement in line 3. However, LIGHTER finds that the only change is removing "-index-1" argument. Consequently, this patch is not detected as a GenProg repair-space patch. This shows that encoding the search space of a repair approach is fundamentally hard, and that a strategy specification may miss some cases.

Answer to RQ3: What is the recall of LIGHTER for repair-space commit detection?
Out of 729 ground-truth cases, we compute that the recall of LIGHTER is 0.92, which is on par or higher than the closest related tools. Per repair approach, the recall has a minimum 0.83 and a median of 0.90, which is consistently high. Therefore, we conclude that LIGHTER can be trusted in terms of detecting commits that actually lie in the search space of program repair approaches. Practitioners can rely on LIGHTER to compute commit coverage on their projects before doing the heavy-duty work of configuring and integrating the actual tool.

Table 7
RQ4: Precision for detected repair-space commits.

Approach	#RSCommit	#TP	Precision
Arja	30	29	0.96
Cardumen	30	25	0.83
Elixir	30	21	0.70
GenProg	30	24	0.80
jMutRepair	7	5	0.71
Kali	30	29	0.96
Nopol	30	18	0.60
NPEfix	30	18	0.60
Total	217	169	0.77

```
- if ((union & 0x0800) == 0) {
+ if ((union & 0x0800) != 0) {
   position.setLatitude(latitude);
   position.setLongitude(longitude);
```

Listing 8: Example of a correctly detected jMutRepair repair-space commits.

6.4. RQ4: Precision of LIGHTER

The computed precision is reported in Table 7. In this table, "#RSCommits" and "#TP" indicate the number of detected repair-space commits in the sample set and the number of true positives, respectively (see Section 5.6 for more details). The precision is computed due to Eq. (2).

Recall that for each repair approach, 30 detected repair-space commits are randomly sampled and manually analyzed.[2] For instance, among the 30 sampled detected repair-space commits for Arja, 29 of them are manually marked as true positives, meaning there are actually potential Arja patches. Therefore, the precision for Arja is 0.96.

We see that LIGHTER has the best precision for Arja and Kali, where only one single commit is wrongly detected as a repair-space commit. In total, 169 out of 217 sampled commits are true positives and the total precision is 77%. We observe that the total precision of LIGHTER is lower than the reported total precision of PPD (91%) (Madeiral et al., 2018), we explain that difference as follows: PPD was fine-tuned for Defects4J, while our experiment considers many more diverse commits and projects.

Listing 8 is an example of a true positive. This commit changes a "==" operator to a "!=" operator and is correctly detected as a jMutRepair repair-space commit. On the other hand, 9 presents an example of a false positive for Nopol. This commit changes the condition of an if statement which in theory is in the search space of Nopol. However, the Nopol manual analyst concluded that the new condition is too complex to be synthesized by Nopol, because Nopol does not support ternary expressions.

Answer to RQ4: What is the precision of LIGHTER for repair-space commit detection?
Thanks to the careful design of the matching criteria, the precision of LIGHTER is 0.77. It is never lower than 0.60 for any of the considered repair approaches. This high precision is important for program repair research: future researchers can rely on LIGHTER to create specifically tailored benchmarks of commits corresponding to the search space of a given repair approach.

2 Except for jMutRepair for which there are only 7 repair-space commits in total.

```
- if (v1.equals(v2)) {
+ if (v1 == null ? v2 == null : v1.equals(v2)) {
   return options.fn();
 }
```

Listing 9: Example of a wrongly detected Nopol repair-space commit. Nopol is not able to synthesize ternary expressions (conditions with "?" and ":" signs).

Table 8
RQ5: Features of Strategy Specifications and Post-matching Rules.

Approach	#Specifications	#Actions	#Entities	LOC
Arja	3	4	5	**343**
Cardumen	3	5	5	273
Elixir	**12**	**17**	**28**	288
GenProg	2	2	3	176
jMutRepair	2	4	6	78
Kali	4	4	4	79
Nopol	4	6	10	275
NPEfix	4	9	24	294
Total	34	51	85	1,806

6.5. RQ5: Complexity of repair search space encoding

The results of the experiment (Section 5.7) are shown in Table 8. Columns "#Specifications", "#Actions", and "#Entities" indicate the total number of strategy specifications, actions and entities for each repair approach that is implemented in LIGHTER. As explained in Section 4.5.1, a specification for a repair strategy outlines the modifications that the strategy performs on the AST nodes of a program. In this context, the modifications are called *actions* and the AST nodes are called *entities* (see Listing 3 as an example of a specification). The "LOC" column of Table 8 shows the number of Java code lines for the post-matching rules implementation. For instance, three strategy specifications are designed to encode the repair strategies employed by Arja. These specifications consist of four actions and five entities in total; the post-matching rules for Arja are implemented in 343 lines of Java code.

In total, we design 34 strategy specifications with 51 actions and 85 entities to encode the search space of all systems. Among all the repair systems, Elixir search space has the most complex encoding specifications with 17 actions and 28 entities. The complexity of Elixir is a consequence of the large number of the different repair strategies that it adopts: for example it includes all "expression update", "statement addition", and "wrap inside if-statement" repair strategies. Moreover, the implementation of post-matching rules contain 1806 lines of code in total. Arja has the largest post-matching rules with 343 lines. The complex post-matching rules for Arja result from the different techniques that it uses to generate a new statement: it can change any literal, variable, or even method of an old statement. This analysis shows that the complexity of specifications and post-matching rules grow as the number of strategies grows in a repair approach.

Finally, one may compare the difficulty of specifying the repair space and running the actual repair systems on past commits. Here is the analysis, with a subjective analysis in parentheses. To run a repair system on past commits, one need: (1) that the system is publicly available (not always the case), (2) that the system can be executed on any commit (uncommon), (3) that the commit can be compiled (hard), (4) that the commit contains a test case (rare). On the contrary, our approach only requires to design strategy specifications and post-matching rules, which is arguably a much more lightweight way of analyzing past commits against repair search spaces.

Answer to RQ5: How complex are the commit matching criteria that encode the repair search space of program repair approaches?
We are able to encode the search space of eight repair approaches using 34 strategy specifications and 1, 806 lines of code. Elixir is the hardest search space to encode, while Kali and jMutRepair are the easiest ones. The biggest advantage of our approach to study the breadth of the search space of program repair is that it is purely static. Consequently, our approach can be considered as lightweight.

6.6. Threats to validity

Complexity of search spaces: Because of the complexity of code change analysis, there is no perfect encoding of repair search space. The encodings implemented in LIGHTER do not yield a perfect matching. There are different factors contributing to false positive and false negatives, incl. noise in the commit, suboptimality of the AST edit script, and corner-cases of the repair approaches not captured in the declarative search space specifications.

Tangled commits: As explained in Section 4.5.2, we consider a commit c as a repair-space commit for approach r only if all the changes in c correspond to a repair strategy employed by r. However, it is known that repositories contain tangled commits where different changes are mixed in the same commit (Herzig and Zeller, 2013). By construction, tangled commits in which only a subset of the commit changes correspond to a repair strategy are not considered as repair-space commits. This contributes to under-estimating the proportion of repair-space commits.

Out-of-file ingredients: As explained in Section 4.9, we consider the ingredients from the "same file" scope. This means the current version of LIGHTER does not detect repair-space commits that use ingredients outside the same file in their patch, such as some patches of jGenProg2/ASTOR when the tool is configured to use the package or application scope. LIGHTER provides the basic framework for considering other ingredient scopes. Researchers and practitioners willing to employ LIGHTER can configure it to consider other scopes than the default one.

Semantically matching commits: LIGHTER considers a commit to be in the search space of an approach only if the changes of the commit are syntactically equivalent to a patch in the search space of that approach. However, there may be many other commits performing changes semantically equivalent to search space patches. By not matching these commits LIGHTER is an underestimation which is sound: practitioners can only be positively surprised about the actual capability of APR approaches.

Dynamic steps of repair: As shown in Table 1, program repair approaches consist of both static and dynamic steps. LIGHTER is focused on those static steps of repair. This means LIGHTER is oblivious to the challenges related to the dynamic steps, such as fault localization. This indicates that the potential of repair approaches computed by LIGHTER is an idealization, and it is by construction higher than what a tool actually delivers.

7. Discussion

7.1. Actionable implications

Here, we summarize the uses of our proposed method for researchers and practitioners.

Prototyping of New Repair Approaches by Researchers: The design space of program repair is very large (Martinez and Monperrus, 2019b) and there are many different potential strategies for generating code fragments and forming patches. There are

```
+ final File head = tempFiles.remove();
+ final String path = head.getPath();
+ final boolean success = head.delete();
+ final File tempFile = tempFiles.removeLast();
+ final String path = tempFile.getPath();
+ final boolean success = tempFile.delete();
```

Listing 10: Commit ca4f6aac in ClassGraph, which renames "head" variable to "tempFile".

also tools that create new repair strategies by mining and analyzing human-written patches (Koyuncu et al., 2018). A number of repair strategies and code synthesis techniques are already implemented in repair tools, such as TBar (Liu et al., 2019b), CapGen (Wen et al., 2018), SimFix (Jiang et al., 2018), etc. These tools let us assess the performance of the particular repair approaches they have implemented. However, there are still many repair strategies left and implementing them for real and assessing their effectiveness with full execution can be very time consuming. Our method and tool can be used by program repair researchers to assess the search space they envision without taking the hard path of fully implementing a tool. More specifically, LIGHTER enables researchers to measure the extent to which human-written commits are covered by the envisioned search space without full execution. It is worth noting that, as described in Section 3, LIGHTER concentrates on the static steps of repair and provides a lightweight method to analyze repair search spaces. To have a comprehensive evaluation of the actual patches delivered by repair tools, the dynamic steps of repair should also be taken into account.

Guideline for researchers: Use LIGHTER to identify promising repair strategies and filter out those that do not cover many real-world commits.

Evaluation of the Potential Value of Using Program Repair by Practitioners: The practitioners who consider automatic repair need to first assess the potential of repair tools on their own projects. Execution of existing tools is an option, but there are two major obstacles to execute them on real world projects. First, configuring a repair tool and actually executing it on a diverse set of projects is hard (Kechagia et al., 2021). For example, TBar (Liu et al., 2019b) is a template-based repair tool whose current version is designed to run experiments only on the Defects4J dataset. It takes extensive work to configure TBar such that it can be run on any arbitrary project (Applelid, 2021). Second, many past bug-fix commits do not come with a failing test case (Madeiral et al., 2019).

Our proposed method can help practitioners assess the potential value of repair tools without facing the mentioned obstacles of full execution. Our contribution allows practitioners to quickly measure how many of their historical changes lie in the search space of repair approaches. This can be done without requiring heavy changes in the target project, configuring of a repair tool or having failing test cases for all past commits. This gives developers an estimation of their own bug-fixes that are in the search space of certain repair tools.

Guideline for practitioners: Use LIGHTER to quickly discard repair tools whose search spaces do not cover many past commits of their software project under consideration.

Table 9
Prevalence of different types of commits in the PB dataset.

Commit Type	#Commits
All commits	55,309
Commits with no source change	28,887 (53%)
Commits with source change	26,422 (47%)
Changes in multiple source files	18,839 (34%)
Changes in exactly one source file	7,583 (13%)
Without strategy instance	3,795 (7%)
With strategy instance	3,788 (6%)
Instance not fully covering the commit	3,041 (5%)
Repair-space commits	**747 (1.35%)**

7.2. Dissection of non repair-space commits

Recall that in our analysis, we consider all types of commits from a repository. We showed in Section 6.1 that a small percentage of commits (1.35%) are repair-space commits. Now, we conduct a study on our PB dataset (see Table 3) to find out why so many commits are not repair-space commits.

As shown in Table 9, 53% (28,887/55,309) of the commits do not contain changes in source files of the application (this also includes commits that only modify code related to test cases). Since all of our considered repair approaches are focused on fixing the source code, these commits lie outside our considered repair space. We note that this is promising for program repair which fixes build configuration files, e.g. BuildMedic (Macho et al., 2018).

Among the 26,422 commits that contain source code changes, 18,839 commits (=34% of all 55,309 commits) contain changes in multiple files. These commits are not considered as repair-space commits, since as described in Section 4.3, LighteR targets commits whose changes are completely covered by a single strategy instance. This leaves us with 7,583 commits whose changes are concentrated in a single Java source file. Out of these 7,583 commits, 3,795 (=7% of 55,309) commits do not contain any strategy instances.

Per our manual analysis on a random sample of 30 commits without strategy instances, they can be divided into three common groups. First, there are commits that only change documentation (i.e., code comments). Second, there are commits that introduce novel code fragments that cannot be generated by any existing repair approach (ex., a new string not used in the program before). Third, there are commits that change parts of the code that cannot be changed by our considered repair strategies (ex., changes in method/class names). Each of these three types of changes can be subject to automation in future repair approaches.

Finally, among the 3,788 commits with changes in a single source file that contain at least one strategy instance, 3,041 (=5% of 55,309) commits have changes that are not fully covered by the detected strategy instance. This result is promising as it shows that many of the commits can at least be partially synthesized (Beyer et al., 2021).

By excluding all the commits that do not lie in the search space of any considered repair approaches, we end up with 747 (1.35% of 55,309) commits that are repair-space commits. These are the commits that can be safely used to study the characteristics of repair search spaces.

7.3. Bug type distribution in the wild

Researchers have studied the distribution of various bug types in real-world projects and proposed automatic techniques for classifying bugs (Thung et al., 2012; Catolino et al., 2019). Understanding the common types of bugs can give us a better insight into developers mistakes that lead to defects in programs. An interesting point of discussion is how repair-space commit detection relates to assessing the prevalence of bug types in the wild.

LighteR looks for correspondence between repair strategies and commits in the wild. Since the repair strategies are designed to repair buggy programs, most of the commits corresponding to them (i.e, repair-space commits) are indeed bug-fix commits, as shown by our results for RQ1 (Section 6.1) demonstrating that 62% of repair-space commits are indeed fixing bugs. A deeper look at them reveals that these bug-fix commits can be categorized into different "bug types" based on their corresponding repair strategies. For example, the repair strategies employed by NPEfix are designed to avoid a `NullPointerException` in the code, which means its corresponding bug type might be a missing null check.

In a sense, our proposed method detects those *types of bug-fix commits* for which there exists an APR repair strategy. The scope of considered bug types is closed. In the contrary, in dedicated studies like Thung et al. (2012) and Catolino et al. (2019), the scope is open, they analyze all possible bug types, and not only those for which an APR repair strategy exists. As future work, LighteR can potentially be extended to label the commits with the bug types they contain, similarly to ADD (Madeiral et al., 2018).

8. Related work

8.1. Analysis of the redundancy assumption

The key assumption behind GenProg is that the patch reuses some code from elsewhere in the program, this is called the redundancy assumption. Previous works have investigated this assumption. Barr et al. (2014) and Martinez et al. (2014) studied the assumption behind GenProg (Le Goues et al., 2012a): patches are synthesized using fragments of code already written in the program under repair. Those works measured the redundancy of a commit: for each commit, the redundancy is the percentage of code introduced that was already introduced by a previous commit. Our approach is different, we verify that a single commit lies in the search space of a repair approach. Note that our post-matching rules also verify the redundancy of the introduced code for the repair actions that are based on the redundancy assumption. For example, the post-matching rule of GenProg verifies whether the statements included in a patch already exist in the buggy program.

8.2. Mining bug-fix patterns from bug datasets

Sobreira et al. (2018) manually analyzed 395 ground-truth patches of Defects4J (Just et al., 2014) buggy programs. They first identified abstractions, called repair patterns, occurring recurrently in patches and involving compositions of repair actions. They identified nine repair patterns from the patches in Defects4J, which span 373 patches of the dataset (94.43%).

Madeiral et al. (2018) presented PPD, a detector of repair patterns in patches. PPD performs source code change analysis at abstract syntax tree level and is able to detect the patterns found in Defects4J. PPD and our work have important differences. First, they focus on a repair patterns that capture human-made changes, while we focus on repair strategies that characterize automated fixes from program repair approaches. Second, our approach checks post-matching rules that are specific to repair approaches (as explained in Section 4.6), while PPD exclusively focuses on analyzing AST changes.

8.3. Mining instances of code changes

There are different works that inspect bug-fix commits and patches with the goal of characterizing the bug-fixing activity.

Pan et al. (2009) built a catalog with 27 bug-fix patterns that they manually identified by inspecting the history of seven open-source Java projects. Then, they built a tool for detecting instances of such bug-fix patterns. They finally reported the frequency of each bug-fix pattern.

Other works have mined Pan's pattern instances from other datasets. Campos and Maia (2017) measured the prevalence of the five most common bug-fix patterns from Pan et al. (2009). For this purpose, they queried the Boa dataset (Dyer et al., 2015) to find how many of the 4, 590, 405 included commits follow each pattern. Islam and Zibran (2020) have mined instances of 21 Pan's pattern from bug-fix commits done on 5 Java systems.

Those works have a different goal than ours. First, they focus on mining instances of change patterns inside commits, while we focus on detecting repair-space commits. Secondly, they only do AST differencing, while we note that AST analysis is insufficient to detect repair-space commits. As we presented in Section 4.6, there are important additional rules that must be verified in order to confirm that a patch can be synthesized by a repair approach.

8.4. Data-driven program repair

Similar to Pan et al. (2009), Kim et al. (2013) manually inspected patches of open-source projects and from that inspection they defined 10 fix templates. Then, they proposed Pattern-based Automatic Program Repair (PAR), a technique that applies these fix templates on faulty programs. Other works have analyzed the presence of PAR's fix templates on bug-fix patches. For example, Soto et al. (2016) detected instances of PAR templates (Kim et al., 2013) from bug-fixes done in Java projects. For that, they analyzed 4,590,679 bug-fixing revisions queried from the Boa platform (Dyer et al., 2015). They found that the most frequent PAR template was "add or remove a branch condition" pattern which appeared in 4.23% of the bug-fixing revisions. We discuss the differences at the end of this subsection.

Martinez and Monperrus (2015) built a probabilistic model of repair actions for guiding the navigation of the search space. For this purpose, they first compute the frequency of particular repair operators (ex., "statement insert of method invocation") in a large dataset of 89,993 real-world commits. This work is different from our study because Martinez and Monperrus check the commits against specific repair operators, while LighteR checks them against search space specifications that include repair strategies (comprising several operators) and post-matching rules.

Soto and Le Goues (2018) also created a probabilistic model of edit distributions that was used by a repair system to repair faster. For that, the authors mined repair operators from bug-fixes done on the 500 most-starred Java projects on Github. They encoded 19 operators in total, selected from those defined by GenProg (Le Goues et al., 2012a), PAR (Kim et al., 2013), SPR (Long and Rinard, 2015) and three additional PAR templates.

Ghanbari et al. (2019) have mined real bug-fix patches from the HDRepair dataset (Le et al., 2016) to measure the frequency of their repair operators implemented in their approach PraPR. Their goal was to further confirm the generality of the 18 PraPR mutators. The PraPR's "*MR mutator*", which mutates method invocation instructions, is the most frequent operator: it appeared in 8.76% of the bug-fix patches for the HDRepair dataset.

Those works and ours do the identification of *instances* of bug-fix patterns. However, none of them identifies repair-space commits. For that, our approach does advanced detection of strategy instances, and also checks rules that are specific to each repair approach. Moreover, none of those papers evaluates the accuracy and precision of their tool as we do in this paper.

8.5. Analysis of the patch search space

Weimer et al. (2013) presented AE, a repair approach that is specifically designed for optimizing the search space, using a cost model and multiple optimizations. For the evaluation of AE, the authors measured the size of the search spaces of AE and GenProg (Le Goues et al., 2012b). Our analysis is different, we do not measure the size of search spaces, we measure the inclusion of real past commits in those search spaces.

Long and Rinard (2016a) presented a systematic analysis of the SPR (Long and Rinard, 2015) and Prophet (Long and Rinard, 2016b) patch search spaces. With respect to our paper, the most related contribution of Long and Rinard (2016a) is that they analyze the density of correct and plausible patches in the search space, and they characterize a trade-off between the size and sophistication of the search space. Our approach has a different goal, we do not analyze plausibility, we analyze past commits from repositories to assess applicability of program repair. More importantly, Long and Rinard run the repair tools to produce the candidate patches in their search space and perform further analysis on them, while we statically specify the repair search space to gain an understanding of their candidate patches.

Petke et al. (2019) have surveyed the literature on the search spaces of genetic improvement, where they consider that program repair is one subset of such search spaces. Our paper provides a novel methodology for studying repair search space, which encodes search spaces with strategy specifications and rules, and it would be helpful for genetic improvement research beyond program repair.

9. Conclusion

In this paper, we have presented an original method for evaluating the breadth of the search space of program repair approaches by analyzing past commits. The key advantage of our approach is that it does not require to configure and execute repair tools on every single commit. Using our approach, we analyze 55,309 human-written commits from 72 Github projects. Our original experiments validate the concept of using static analysis in order to study the breadth of the search space of program repair approaches.

CRediT authorship contribution statement

Khashayar Etemadi: Methodology, Software, Investigation, Data curation, Writing – original draft. **Niloofar Tarighat:** Software, Investigation, Data curation. **Siddharth Yadav:** Software, Investigation, Data curation, Writing – original draft. **Matias Martinez:** Conceptualization, Methodology, Writing – review & editing, Supervision. **Martin Monperrus:** Conceptualization, Methodology, Writing – review & editing, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, Sweden.

References

Aleti, A., Martinez, M., 2021. E-APR: mapping the effectiveness of automated program repair techniques. Empir. Softw. Eng. 26 (5), 1–30.

Applelid, G., 2021. Evaluating template-based automatic program repair in industry. (Master's thesis). In: TRITA-EECS-EX 2021:697, KTH, School of Electrical Engineering and Computer Science (EECS), p. 103.

Barr, E.T., Brun, Y., Devanbu, P., Harman, M., Sarro, F., 2014. The plastic surgery hypothesis. In: Proceedings Of The 22nd ACM SIGSOFT International Symposium On Foundations Of Software Engineering. In: FSE 2014, Association for Computing Machinery, New York, NY, USA, pp. 306–317. http://dx.doi.org/10.1145/2635868.2635898.

Beyer, D., Grunske, L., Lemberger, T., Tang, M., 2021. Towards a benchmark set for program repair based on partial fixes. arXiv preprint arXiv:2107.08038.

Campos, E.C., Maia, M.A., 2017. Common bug-fix patterns: A large-scale observational study. In: Proceedings Of The 11th ACM/IEEE International Symposium On Empirical Software Engineering And Measurement. In: ESEM '17, IEEE Press, pp. 404–413. http://dx.doi.org/10.1109/ESEM.2017.55.

Catolino, G., Palomba, F., Zaidman, A., Ferrucci, F., 2019. Not all bugs are the same: Understanding, characterizing, and classifying bug types. J. Syst. Softw. 152, 165–181.

Cornu, B., Durieux, T., Seinturier, L., Monperrus, M., 2015. Npefix: Automatic runtime repair of null pointer exceptions in java. arXiv preprint arXiv:1512.07423.

Durieux, T., Cornu, B., Seinturier, L., Monperrus, M., 2017. Dynamic patch generation for null pointer exceptions using metaprogramming. In: 2017 IEEE 24th International Conference On Software Analysis, Evolution And Reengineering (SANER). IEEE, pp. 349–358.

Durieux, T., Madeiral, F., Martinez, M., Abreu, R., Empirical review of Java program repair tools: a large-scale experiment on 2141 bugs and 23, 551 repair attempts. In: Proceedings Of The 2019 27th ACM Joint Meeting On European Software Engineering Conference And Symposium On The Foundations Of Software Engineering, pp. 302–313.

Durieux, T., Monperrus, M., 2016. Dynamoth: dynamic code synthesis for automatic program repair. In: Proceedings Of The 11th International Workshop On Automation Of Software Test, pp. 85–91.

Dyer, R., Nguyen, H.A., Rajan, H., Nguyen, T.N., 2015. Boa: Ultra-large-scale software repository and source-code mining. ACM Trans. Softw. Eng. Methodol. (TOSEM) 25 (1), 1–34.

Etemadi, K., Tarighat, N., Yadav, S., Martinez, M., Monperrus, M., 2021. Lighter tool and dataset. https://github.com/khaes-kth/GithubRepairPatterns.

Falleri, J.-R., Morandat, F., Blanc, X., Martinez, M., Monperrus, M., 2014. Fine-grained and accurate source code differencing. In: Proceedings Of The 29th ACM/IEEE International Conference On Automated Software Engineering, pp. 313–324.

Ghanbari, A., Benton, S., Zhang, L., 2019. Practical program repair via bytecode mutation. In: Proceedings Of The 28th ACM SIGSOFT International Symposium On Software Testing And Analysis. In: ISSTA 2019, Association for Computing Machinery, New York, NY, USA, pp. 19–30. http://dx.doi.org/10.1145/3293882.3330559.

Goues, C.L., Brun, Y., Forrest, S., Weimer, W., 2017. Clarifications on the construction and use of the ManyBugs benchmark. IEEE Trans. Software Eng. 43 (11), 1089–1090. http://dx.doi.org/10.1109/TSE.2017.2755651.

Herzig, K., Zeller, A., 2013. The impact of tangled code changes. In: 2013 10th Working Conference On Mining Software Repositories (MSR). IEEE, pp. 121–130.

Islam, M.R., Zibran, M.F., 2020. How bugs are fixed: exposing bug-fix patterns with edits and nesting levels. In: Proceedings Of The 35th Annual ACM Symposium On Applied Computing, pp. 1523–1531.

Jiang, J., Xiong, Y., Zhang, H., Gao, Q., Chen, X., 2018. Shaping program repair space with existing patches and similar code. In: Proceedings Of The 27th ACM SIGSOFT International Symposium On Software Testing And Analysis, pp. 298–309.

Just, R., Jalali, D., Ernst, M.D., 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In: Proceedings Of The 2014 International Symposium On Software Testing And Analysis, pp. 437–440.

Kechagia, M., Mechtaev, S., Sarro, F., Harman, M., 2021. Evaluating automatic program repair capabilities to repair API misuses. IEEE Trans. Softw. Eng..

Kim, D., Nam, J., Song, J., Kim, S., 2013. Automatic patch generation learned from human-written patches. In: 2013 35th International Conference On Software Engineering (ICSE). IEEE, pp. 802–811.

Koyuncu, A., Liu, K., Bissyandé, T.F., Kim, D., Klein, J., Monperrus, M., Traon, Y.L., 2018. Fixminer: Mining relevant fix patterns for automated program repair. arXiv preprint arXiv:1810.01791.

Le, X.D., Bao, L., Lo, D., Xia, X., Li, S., 2019. On reliability of patch correctness assessment. In: Proceedings Of The 41st ACM/IEEE International Conference On Software Engineering.

Le, X.B.D., Lo, D., Goues, C.L., 2016. History driven program repair. In: 2016 IEEE 23rd International Conference On Software Analysis, Evolution, And Reengineering (SANER), vol. 1, pp. 213–224.

Le, X.B.D., Thung, F., Lo, D., Le Goues, C., 2018. Overfitting in semantics-based automated program repair. Empir. Softw. Eng. 1–27.

Le Goues, C., Dewey-Vogt, M., Forrest, S., Weimer, W., 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In: Proceedings Of The International Conference On Software Engineering, pp. 3–13.

Le Goues, C., Nguyen, T., Forrest, S., Weimer, W., 2012a. GenProg: A Generic method for automatic software repair. IEEE Trans. Softw. Eng. 38 (1), 54–72.

Lin, B., Wang, S., Wen, M., Zhang, Z., Wu, H., Qint, Y., Mao, X., 2020. Understanding the non-repairability factors of automated program repair techniques. In: 2020 27th Asia-Pacific Software Engineering Conference (APSEC). IEEE, pp. 71–80.

Liu, K., Koyuncu, A., Bissyandé, T.F., Kim, D., Klein, J., Le Traon, Y., 2019a. You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In: 2019 12th IEEE Conference On Software Testing, Validation And Verification (ICST). IEEE, pp. 102–113.

Liu, K., Koyuncu, A., Kim, D., Bissyandé, T.F., 2019. Tbar: Revisiting template-based automated program repair. In: Proceedings Of The 28th ACM SIGSOFT International Symposium On Software Testing And Analysis, pp. 31–42.

Liu, K., Wang, S., Koyuncu, A., Kim, K., Bissyande, T.F.D.A., Kim, D., Wu, P., Klein, J., Mao, X., Le Traon, Y., 2020. On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs. In: 42nd ACM/IEEE International Conference On Software Engineering (ICSE).

Long, F., Rinard, M., 2015. Staged program repair with condition synthesis. In: Proceedings Of The 2015 10th Joint Meeting On Foundations Of Software Engineering. In: ESEC/FSE 2015, Association for Computing Machinery, New York, NY, USA, pp. 166–178. http://dx.doi.org/10.1145/2786805.2786811.

Long, F., Rinard, M., 2016a. An analysis of the search spaces for generate and validate patch generation systems. In: Proceedings Of The 38th International Conference On Software Engineering. In: ICSE '16, Association for Computing Machinery, New York, NY, USA, pp. 702–713. http://dx.doi.org/10.1145/2884781.2884872.

Long, F., Rinard, M., 2016b. Automatic patch generation by learning correct code. SIGPLAN Not 51 (1), 298–312. http://dx.doi.org/10.1145/2914770.2837617.

Macho, C., McIntosh, S., Pinzger, M., 2018. Automatically repairing dependency-related build breakage. In: 2018 IEEE 25th International Conference On Software Analysis, Evolution And Reengineering (SANER). IEEE, pp. 106–117.

Madeiral, F., Durieux, T., Sobreira, V., Maia, M., 2018. Towards an automated approach for bug fix pattern detection. In: VEM '18 - Proceedings Of The VI Workshop On Software Visualization, Evolution And Maintenance. São Carlos, Brazil, URL https://hal.archives-ouvertes.fr/hal-01851813.

Madeiral, F., Urli, S., Maia, M., Monperrus, M., 2019. Bears: An extensible java bug benchmark for automatic program repair studies. In: 2019 IEEE 26th International Conference On Software Analysis, Evolution And Reengineering (SANER). IEEE, pp. 468–478.

Martinez, M., Durieux, T., Sommerard, R., Xuan, J., Monperrus, M., 2017. Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. Empir. Softw. Eng. 22 (4), 1936–1964.

Martinez, M., Monperrus, M., 2015. Mining software repair models for reasoning on the search space of automated program fixing. Empir. Softw. Eng. 20 (1), 176–205.

Martinez, M., Monperrus, M., 2016. Astor: A program repair library for java. In: Proceedings Of The 25th International Symposium On Software Testing And Analysis, pp. 441–444.

Martinez, M., Monperrus, M., 2018. Ultra-large repair search space with automatically mined templates: The cardumen mode of astor. In: International Symposium On Search Based Software Engineering. Springer, pp. 65–86.

Martinez, M., Monperrus, M., 2019. Coming: a tool for mining change pattern instances from git commits. In: 2019 IEEE/ACM 41st International Conference On Software Engineering: Companion Proceedings (ICSE-Companion). IEEE, pp. 79–82.

Martinez, M., Monperrus, M., 2019a. Astor: Exploring the design space of generate-and-validate program repair beyond GenProg. J. Syst. Softw. Elsevier.

Martinez, M., Monperrus, M., 2019b. Astor: Exploring the design space of generate-and-validate program repair beyond GenProg. J. Syst. Softw. Elsevier http://dx.doi.org/10.1016/j.jss.2019.01.069, URL http://arxiv.org/pdf/1802.03365.

Martinez, M., Weimer, W., Monperrus, M., 2014. Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches. In: Companion Proceedings Of The 36th International Conference On Software Engineering, pp. 492–495.

Monperrus, M., 2017. Automatic software repair: a bibliography. ACM Comput. Surv. 51, 1–24. http://dx.doi.org/10.1145/3105906, URL https://hal.archives-ouvertes.fr/hal-01206501/file/survey-automatic-repair.pdf.

Murphy-Hill, E.R., Zimmermann, T., Bird, C., Nagappan, N., 2013. The design of bug fixes. In: Notkin, D., Cheng, B.H.C., Pohl, K. (Eds.), 35th International Conference On Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013. IEEE Computer Society, pp. 332–341. http://dx.doi.org/10.1109/ICSE.2013.6606579.

Pan, K., Kim, S., Whitehead, E.J., 2009. Toward an understanding of bug fix patterns. Empir. Softw. Eng. 14 (3), 286–315.

Petke, J., Alexander, B., Barr, E.T., Brownlee, A.E.I., Wagner, M., White, D.R., 2019. A survey of genetic improvement search spaces. In: Proceedings Of The Genetic And Evolutionary Computation Conference Companion. In: GECCO '19, Association for Computing Machinery, New York, NY, USA, pp. 1715–1721. http://dx.doi.org/10.1145/3319619.3326870.

Qi, Z., Long, F., Achour, S., Rinard, M., 2015. 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, pp. 24–36.

Qin, Y., Wang, S., Liu, K., Mao, X., Bissyandé, T.F., 2021. On the impact of flaky tests in automated program repair. In: 2021 IEEE International Conference On Software Analysis, Evolution And Reengineering (SANER). IEEE, pp. 295–306.

Saha, R.K., Lyu, Y., Lam, W., Yoshida, H., Prasad, M.R., 2018. Bugs. jar: a large-scale, diverse dataset of real-world java bugs. In: Proceedings Of The 15th International Conference On Mining Software Repositories, pp. 10–13.

Saha, R.K., Lyu, Y., Yoshida, H., Prasad, M.R., 2017. Elixir: Effective object-oriented program repair. In: 2017 32nd IEEE/ACM International Conference On Automated Software Engineering (ASE). IEEE, pp. 648–659.

Sobreira, V., Durieux, T., Madeiral, F., Monperrus, M., Maia, M.A., 2018. Dissection of a bug dataset: Anatomy of 395 patches from Defects4J. In: Proceedings Of SANER.

Soto, M., Le Goues, C., 2018. Using a probabilistic model to predict bug fixes. In: 2018 IEEE 25th International Conference On Software Analysis, Evolution And Reengineering (SANER), pp. 221–231.

Soto, M., Thung, F., Wong, C.-P., Le Goues, C., Lo, D., 2016. A deeper look into bug fixes: Patterns, replacements, deletions, and additions. In: Proceedings Of The 13th International Conference On Mining Software Repositories. In: MSR '16, Association for Computing Machinery, New York, NY, USA, pp. 512–515. http://dx.doi.org/10.1145/2901739.2903495.

Thung, F., Lo, D., Jiang, L., 2012. Automatic defect categorization. In: 2012 19th Working Conference On Reverse Engineering. IEEE, pp. 205–214.

Tufano, M., Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., Poshyvanyk, D., 2017. There and back again: Can you compile that snapshot? J. Softw. Evol. Process 29 (4), e1838.

Weimer, W., Fry, Z.P., Forrest, S., 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In: Proceedings Of The 28th IEEE/ACM International Conference On Automated Software Engineering, pp. 356–366.

Weimer, W., Nguyen, T., Le Goues, C., Forrest, S., 2009. Automatically finding patches using genetic programming. In: 2009 IEEE 31st International Conference On Software Engineering. IEEE, pp. 364–374.

Wen, M., Chen, J., Wu, R., Hao, D., Cheung, S.-C., 2018. Context-aware patch generation for better automated program repair. In: 2018 IEEE/ACM 40th International Conference On Software Engineering (ICSE). IEEE, pp. 1–11.

White, M., Tufano, M., Martinez, M., Monperrus, M., Poshyvanyk, D., 2019. Sorting and transforming program repair ingredients via deep learning code similarities. In: 2019 IEEE 26th International Conference On Software Analysis, Evolution And Reengineering (SANER). IEEE, pp. 479–490.

Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F., 2016. A survey on software fault localization. IEEE Trans. Softw. Eng. 42 (8), 707–740.

Xiong, Y., Liu, X., Zeng, M., Zhang, L., Huang, G., 2018. Identifying patch correctness in test-based program repair. In: Proceedings Of The 40th International Conference On Software Engineering. pp. 789–799.

Xuan, J., Martinez, M., Demarco, F., Clement, M., Marcote, S.L., Durieux, T., Le Berre, D., Monperrus, M., 2016. Nopol: Automatic repair of conditional statement bugs in java programs. IEEE Trans. Softw. Eng. 43 (1), 34–55.

Yang, D., Liu, K., Kim, D., Koyuncu, A., Kim, K., Tian, H., Lei, Y., Mao, X., Klein, J., Bissyandé, T.F., 2021. Where were the repair ingredients for Defects4j bugs? Empir. Softw. Eng. 26 (6), 1–33.

Ye, H., Gu, J., Martinez, M., Durieux, T., Monperrus, M., 2021b. Automated classification of overfitting patches with statically extracted code features. IEEE Trans. Softw. Eng. http://dx.doi.org/10.1109/tse.2021.3071750, URL http://arxiv.org/pdf/1910.12057.

Ye, H., Martinez, M., Monperrus, M., 2021a. Automated patch assessment for program repair at scale. Empir. Softw. Eng. 26 (2), 1–38.

Yuan, Y., Banzhaf, W., 2018. ARJA: AUtomated repair of java programs via multi-objective genetic programming. IEEE Trans. Softw. Eng..

Khashayar Etemadi is a Ph.D. student at KTH Royal Institute of Technology, Stockholm, 11428, Sweden. His research focuses on explainable software robots. Etemadi received his M.Sc. in software engineering from Sharif University of Technology, Tehran, Iran. Contact him at khaes@kth.se.

Niloofar Tarighat is a M.Sc. student at McGill University, Canada, working on machine learning and computer vision. You can contact her at tarighat_niloofar@ee.sharif.edu.

Siddharth Yadav is a DevOps engineer at Zeta Suite. He received his B.Sc. from Indraprastha Institute of Information Technology Delhi, India. You can contact him at siddharth16268@iiitd.ac.in.

Matias Martinez is an associate professor at Université Polytechnique Hauts-de-France, Valenciennes, 59260, France. His research focuses on automated software engineering, software testing, and programming languages. Martinez received his Ph.D. from the University of Lille, France. Further information about him can be found at http://www.martinezmatias.com/. Contact him at matias.martinez@uphf.fr.

Martin Monperrus is a professor of software technology at KTH Royal Institute of Technology, Stockholm, 10044, Sweden. His main research interest lies in the field of software engineering, with a current focus on automatic program repair, program hardening, and chaos engineering. Monperrus received his Ph.D. from the University of Rennes 1. Further information about him can be found at https://www.monperrus.net/martin/. Contact him at monperrus@kth.se.