The Journal of Systems and Software 151 (2019) 65–80

Contents lists available at ScienceDirect

The Journal of Systems and Software

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

Astor: Exploring the design space of generate-and-validate program repair beyond GenProg

Matias Martinez[a,*], Martin Monperrus[b]

[a] Université Polytechnique Hauts-de-France, France
[b] KTH Royal Institute of Technology, Sweden

ARTICLE INFO

Article history:
Received 10 February 2018
Revised 7 January 2019
Accepted 22 January 2019
Available online 1 February 2019

Keywords:
Software Maintenance
Automated Program Repair
Software Testing
Evaluation Frameworks
Software Bugs
Defects

ABSTRACT

This article contributes to defining the design space of program repair. Repair approaches can be loosely characterized according to the main design philosophy, in particular "generate- and-validate" and synthesis-based approaches. Each of those repair approaches is a point in the design space of program repair. Our goal is to facilitate the design, development and evaluation of repair approaches by providing a framework that: a) contains components commonly present in most approaches, b) provides built-in implementations of existing repair approaches. This paper presents a Java framework named Astor that focuses on the design space of generate-and-validate repair approaches. The key novelty of Astor is to provides explicit extension points to explore the design space of program repair. Thanks to those extension points, researchers can both reuse existing program repair components and implement new ones. Astor includes 6 unique implementations of repair approaches in Java, including GenProg for Java called jGenProg. Researchers have already defined new approaches over Astor. The implementations of program repair approaches built already available in Astor are capable of repairing, in total, 98 real bugs from 5 large Java programs. Astor code is publicly available on Github: https://github.com/SpoonLabs/astor.

1. Introduction

Automated software repair is a research field that has emerged during the last decade for repairing real bugs of software application. The main goal is to reduce cost and time of software maintenance by proposing to developers automatically synthesized patches that solve bugs present in their applications. Among pioneer repair systems are GenProg (Weimer et al., 2009), Semfix (Duong et al., 2013), Prophet (Long and Rinard, 2016), Nopol (Xuan et al., 2017), and others (Kim et al., 2013; Xiong et al., 2017; Mechtaev et al., 2016; Le et al., 2017a,b; Ke et al., 2015; Long and Rinard, 2016; Long et al., 2017; Perkins et al., 2009; Durieux et al., 2017a; Durieux and Monperrus, 2016; Weimer et al., 2013; Mechtaev et al., 2015; Qi et al., 2014; Long and Rinard, 2015). Automation of bug fixing is possible by using *automated correctness oracles*. For instance, GenProg (Weimer et al., 2009) introduced the use of *test suite* as correctness oracle: the correctness of a bug fix is is assessed by executing all tests from its associated test suite.

Program repair systems can be loosely characterized along their main design philosophy: generate-and-validate approaches (which first generate a set of candidate patches, and then validate them against the test suite) or synthesis based approaches (which first use test execution information to build a repair constraint, and then use a constraint solver to synthesize a patch). For example, GenProg and JAFF (Arcuri, 2011) are generate-and-validate approaches based on *genetic programming*.

More generally, every repair system is a point in the design space of program repair. By making design decisions explicit in that design space, one can start to have a fine-grain understanding of the core conceptual differences in the field. For example, the main conceptual difference between GenProg and PAR (Kim et al., 2013) lies in the repair operators: they do not use the same code transformations for synthesizing patches.

To foster research on program repair, we aim at providing the research community with a generic framework that encodes the code design space of generate-and-validate repair approaches.

In this paper, our main contribution is Astor (Automatic Software Trans-formations fOr program Repair). Astor is a program repair framework for Java, it provides 6 *generate and validate* repair approaches: jGenProg (a Java implementation of GenProg, originally in C), jKali (an implementation of Kali Qi et al., 2015a, originally in C), jMutRepair (an implementation of MutRepair Debroy and Wong, 2010, not publicly available), DeepRepair (White et al., 2017), Cardumen (Martinez and Monperrus, 2018), and TIBRA (an extension of jGenProg introduced in this paper).

* Corresponding author.
E-mail address: matias.martinez@univ-valenciennes.fr (M. Martinez).

https://doi.org/10.1016/j.jss.2019.01.069
0164-1212/

Those repair approaches are based on twelve extension points that form the first ever explicit design space of program repair. Over those twelve extension points, the program repair researchers can both choose an existing component (among 33 ones), or implement new ones for exploring a new point in the design space of program repair.

Astor has been extensively used by the research community
a) for creating a novel repair system based on Astor (Tanikado et al., 2017; White et al., 2017; Yu et al., 2017; Wen et al., 2017), which is the key enabling factor,
b) for performing comparative evaluations, using Astor's publicly available implementation of existing approaches (Xin and Reiss, 2017b),
c) for reusing numerical results and/or patches obtained with Astor (Xiong et al., 2017; Le et al., 2016; Saha et al., 2017; Yuan and Banzhaf, 2017).

Astor is publicly available on Github and is actively maintained. A user community is able to provide support. Bug fixes and extensions are welcome as external contributions (pull requests). From an open-science perspective, since the whole code base is public, peer researchers can validate the correctness of the implementation and hence minimize threats to internal validity.

To sum up, our contributions are:

- The explicit design space of generate-and-validate program repair.
- The realization of that design space in Astor, where the most important design decisions of program repair are encoded as extension point.
- Twelve extension points, which program repair researchers can either reuse or extend for doing interesting research in the field.
- Six repair approaches that can be used out-of-the-box in comparative evaluations, incl. jGenProg, the most used Java implementation of GenProg according to citation impact (Weimer et al., 2009; Goues et al., 2012b,a; Weimer et al., 2010; Forrest et al., 2009). The study of the repair capability of the implemented approaches based on the Defects4J bug benchmark (Just et al., 2014).
- The evaluation of two extension points of GenProg: choice of the ingredient space and ingredient transformation.

This paper is a completely rewritten long version of a short paper (Martinez and Monperrus, 2016). It includes a detailed explanation of Astor's architecture, extension pointss as well as a large evaluation. The paper continues as follows. Section 2 describes the design of Astor, Section 3 presents the extension points provided by Astor. Section 4 presents the built-in approaches included in Astor. Section 5 presents a evaluation of the built-in approaches and different implementations for the extension points. Section 6 presents the related work. Finally, section 8 concludes the paper.

2. Architecture

2.1. The Design of Astor

Astor is a framework that allows researchers to implement new automated program repair approaches, and to extend available repair approaches such as jGenProg (Martinez et al., 2016) (implementation of GenProg Weimer et al., 2009), jKali (Martinez and Monperrus, 2016) (implementation of Kali (Qi et al., 2015b)), jMutRepair (Martinez and Monperrus, 2016) (implementation from MutRepair (Debroy and Wong, 2010)), DeepRepair (White et al., 2017), Cardumen (Martinez and Monperrus, 2018).

Astor encodes the design space of *generate-and-validate* repair approaches, which first search within a search space to generate a set of patches, and then validate using a correctness oracle. Astor provides twelve extension points that form the design space of generate-and-validate program repair. New approaches can be implemented by choosing an existing component for each extension point, or to implement new ones.

The extension points allow Astor users to define the design of a repair approach. Main design decisions are: *a)* code transformations (aka repair operators) used to define the solution search space; *b)* different strategies for navigating the search space of candidate solutions; and *c)* mechanism for validating a candidate solution.

Astor was originally conceived for building *test-suite based repair approaches* (Weimer et al., 2009) and the first implemented approach over it was named jGenProg, a Java implementation of GenProg (Weimer et al., 2009), originally written in OCaml language for repairing C code. In test-suite based repair, test suites are considered as a proxy to the program specification, and a program is considered as fulfilling its specification if its test suite passes all the these cases otherwise, the program has a defect. The test suite is used as a *bug oracle*, i.e., it asserts the presence of the bug, and as *correctness oracle*.

An approach over Astor requires as input a buggy program to be repaired and a correctness oracle such as a test suite. As output, the approach generates, when it is possible, one or more patches that are valid according to the correctness oracle.

Algorithm 1 displays the high-level steps executed, in sequence, by a *generate-and-validate* repair approach built in Astor. They are: *1)* Fault localization (line 1), *2)* Creation of a representation (line 2), *3)* Navigation of the search space (lines 7-11), and *4)* Solution post-processing (line 12). In the remainder of this section, we describe each step.

Algorithm 1 Main steps of *generate-and-validate* repair approaches, implemented in Astor (extension points are referred to as comment prefixed by //,)

Require: Program under repair *P*
Require: test suite *TS*
Ensure: A list of test-suite adequate patches
1: suspicious ← run-fault-localization(P, TS) //EP_FL
2: mpl ← create-modification-points(suspicious) //EP_MPG
3: ops ← get-operators() //EP_OD
4: tsa-patches-refined ← ∅
5: nr-iteration ← 0
6: starting-time ← System.currentTime
7: **while** continue-searching(starting-time, nr-iteration, size(tsa-patches)) **do**
8: program-variants ← generate-program-variants(P, mpl, ops)
9: tsa-patches ← tsa-patches + validate-variants(P, program-variants, TS)
10: nr-iteration ← nr-iteration + 1
11: **end while**//EP_NS
12: tsa-patches-refined ← refining-patches(tsa-patches) //EP_SP
13: **return** tsa-patches-refined

2.2. Fault Localization

The *fault localization* step is the first step executed by an approach built over Astor. It aims at determining what is wrong in the program received as input. This step is executed on line 1 of

Algorithm 1. Fault localization consists of computing locations that are suspicious. In the context of repair, fault localization allows to reduce the search space by discarding those code locations that are probably healthy. A repair approach can use the suspiciousness values of locations to guide the search in the solution space. Consequently, fault localization has an impact on the effectiveness of the repair approach (Qi et al., 2013b).

Test-suite based repair approaches from Astor use fault localization techniques based on spectrum analysis. Those techniques execute the test cases of a buggy program and trace the execution of software components (e.g., methods, lines). Then, from the collected traces and the tests results (i.e., fail or pass), the techniques use formulas to calculate the suspicious value of each component. The suspicious value goes for 0 (lowest probability that the component contains a bug) to 1 (highest). Repair approaches use different formulas, for instance, GenProg uses an ad-hoc formula (Weimer et al., 2009), while MutRepair (Debroy and Wong, 2010) uses the Tarantula formula (Jones et al., 2002).

Astor provides fault localization as an extension point named EP_FL, where researchers can plug implementations of any fault localization technique. Astor provides a component (used by default) that implement that point and uses the fault localization library named GZoltar (Campos et al., 2012) and the Ochiai formula (Abreu et al., 2006).[1]

2.3. Identification of Modification Points

Once the fault localization step returns a list of suspicious code locations (such as statements), Astor create a representation of the program under repair.

> Definition 1: A *Modification point* is a code element (e.g., a statement, an expression) from the buggy program under repair that can be modified with the goal of repairing the bug.

Astor creates modification points from the suspicious statements returned by the fault localization step (Section 2.2). This step is executed in line 2 of Algorithm 1. Astor provides an extension point named EP_MPG (Section 3.2) to define the granularity of each modification point according to that one targeted by a repair approach built over Astor. For instance, jGenProg creates one modification point per each statement indicated as suspicious by the fault localization. Cardumen, another approach built over Astor, works at a fine-grained level: it creates a modification point for each expression contained in a suspicious statement. Other approaches focus on particular code elements, such as jMutRepair which creates modification points only for expressions with unary and binary operators. For example, let us imagine that the fault localization marks as suspicious the two lines presented in Listing 1

```
9       ....
10      myAccount = getAccount(name);
11      myAccount.setBalance(previousMonth + currentMonth);
12      ....
```

Listing 1. Two suspicious statements.

.

jGenProg creates two modification points, both pointing to statements, one to the assignment at line 10, another to the method invocation at line 11. Contrary, Cardumen creates 3 modifications points: one pointing to the expression at the right size of the assignment at line 10, a second one to the method invocation at line 11 (note that the method invocation it is also an expression), and the last one pointing to the expression (`previousMonth + currentMonth`) which is the parameter of the method invocation at line 11.

2.4. Creation of repair operators

Astor synthesizes patches by applying automated code transformation over suspicious modification points. Those transformations are done by *repair operators* and the set of all repair operators that an approach considers during the repair conform the *repair operator space*.

> Definition 2: a *Repair operator* is an action that transforms a code element associated to a modification point into another, modified compatible code element.

Astor provides an extension point named EP_OD (Section 3.5) for specifying the operator space that a repair approach will use. The extension point is invoked at line 3 of Algorithm 1. Astor works with two kinds of repair operators:

Synthesis and repair operators. An approach can synthesize new code by directly applying one transformation operators to a modification point, without the need of any extra information (in particular without using ingredients). One of them is the repair operator from jMutRepair which changes a logical operator from $>$ to $>=$. For example, it generates the new code `(fa * fb) >= 0.0` from the code `(fa * fb) > 0.0` without the use of any further information.

Synthesis based on ingredients. There are operators that need some extra information before applying a code transformation in a modification point mp_i. For instance, two operators (Insert and Replace) from GenProg (Weimer et al., 2009) need one statement (aka the *ingredient*) taken from somewhere in the application under repair. Once selected, the ingredient is inserted before or replace the code at mp_i. Such approaches are known as *Ingredient-based repair approaches* (Weimer et al., 2009). jGenProg and DeepRepair (White et al., 2017) are two ingredient-based approaches built over Astor. Astor gives support to such repair approaches by automatically providing: *a)* a pool of ingredients available, *b)* strategies for selecting ingredients from the pool.

2.5. Navigation of the search space

Once that that all code transformations to be applied to a suspicious element are known, Astor proceeds with the navigation of the search space. The goal is to find, between all possible modified versions of the buggy program, one or more versions that do not contain the bug under repair and that do not introduce new bugs.

The navigation of the search space is executed by the main loop (Algorithm 1 line 7). In each iteration, the approach verifies whether a set of code changes done by repair operators over some modification points (which produce a modified version of the program) repair the bug. A *modified* version of a buggy program is called in Astor a *Program variant*.

> Definition 3: a *Program variant* is a entity that stores: *a)* one or more code locations known as modification point; *b)* the repair operators applied to each modification point; *c)* the code source resulting from the execution of all repair operators over the corresponding modification points.

[1] http://www.gzoltar.com/.

Then, Astor computes a candidate *patch* from a program variant. A *Patch* produced by Astor is a set of changes between the buggy version and a modified version represented by a program variant.

Astor allows to override the navigation strategy using the extension point EP_NS (Section 3.3). Algorithm 1 presents the default implementation of the navigation strategy, named *Selective*, which executes the two main steps that characterize a *generate-and-validate* technique: first, the algorithm generates 1+ program variants (Line), then it validates them against the test suite (Line).

2.5.1. Generation of Program Variants

Algorithm 2 shows the main steps that Astor executes for creating a program variant. Let us analyze each of them.

Algorithm 2 Method *generate-program-variants* (as comment prefixed by //, the name of the extension point)

```
Require: Program under repair P
Require: List of modification points MPs
Require: List of operator OS
1: mps ← choose-modification-points(MPs) //EP_MPS
2: transformations ← ∅
3: for all mp-i ∈ mps do
4:     op-j← choose-operator(mp-i, OS) //EP_OS
5:     transformations ← transformations ∪ create-transformation(P, mp-i, op-j)
6: end for
7: program-variants ← apply-transformations(transformations, P)
8: return program-variants
```

Selection of modification points. First, a repair approach built over Astor chooses, according to a selection strategy, the modifications points (at least one) to apply repair operators (Algorithm 2 line 1). Astor provides an extension point named EP_MPS (Section 3.4) for specifying customized strategies of modification points selection.

Selection of repair operators. For each selected modification point mp_i, an approach selects one repair operator op_j to apply at mp_i (Algorithm 2 line 4) and adds the transformation created by *create-transformation* (Algorithm 3) to the set of code transformations (Algorithm 2 line 5). Astor provides an extension point named EP_OS (Section 3.6) for specifying customized strategies of operator selection.

Algorithm 3 Method *create-transformation* (as comment prefixed by //, the name of the extension point)

```
Require: Program under repair P
Require: Modification point MP
Require: Repair operator OP
1: if needs-ingredient(OP) then
2:     if pool-not-initialized(ingredient-pool) then
3:         ingredient-pool ← build-pool(P) //EP_IPD
4:     end if
5:     ingredient ← select-ingredient(ingredient-pool) // EP_IS
6:     transformed-ingredient ← transform-ingredient(ingredient) //EP_IT
7:     return (MP, OP, transformed-ingredient)
8: else
9:     return (MP, OP)
10: end if
```

Creation of code transformation. In Astor, a code transformation is a concept that groups a modification point *mp* and a repair operator *op*. Algorithm 3 shows the creation of code transformations. When the operator is of kind ingredient-based (Section 2.4) the transformation needs an additional element: an ingredient for synthesizing a patch. Astor provides to those operators a pool that contains all the ingredients that they can use.

For creating a transformation, Astor first detects if the operator needs ingredients or not (Algorithm 3 line 1). If it does not need any ingredient, Astor returns the transformation composed by the *mp* and *op* (Algorithm 3 line 9). Otherwise, Astor creates an *ingredient pool* from the program under repair. The creation involves to first parse the code at a given granularity (by default, that one given by the extension point EP_MPG) and then store each parsed element in the pool (Algorithm 3 line 3). Astor provides an extension point named EP_IPD (Section 3.7) for plugging a customized strategy for building the ingredient pool.

The ingredient pool is queried by the repair approach when an ingredient-based repair operator needs an ingredient for synthesizing the candidate patch code (Algorithm 3 line 5). Astor provides an extension point named EP_IS (section 3.8) for plugging in a customized strategy in order to select an ingredient from the ingredient pool.

Moreover, when an operator gets an ingredient for the ingredient pool, it can use it directly (i.e., without applying any transformation) or after applying a transformation over the ingredient (Algorithm 3 line 6). For instance, a transformation proposed by Astor is to replace variables from the ingredient that is not in the scope of the modification point. Astor provides an extension point named EP_IT (section 3.9) for plugging a customized strategy of ingredients transformation.

Creation of program variants. A repair approach over Astor generates program variants from the code transformation previously generated (Algorithm 2 line 7) and returns them for validation stage (line 8).

2.5.2. Candidate patch validation

Algorithm 4 shows the main steps that Astor executes for validating program variants and returning test-suite adequate patches. Once program variants are created, the Astor framework synthesizes from each variant the code source of the patch (Algorithm 4 line 2), then applies it to the buggy version of the program under repair and finally evaluates the modified version (line 3) using the correctness oracle. If the patched version is valid, the corresponding patch is a solution and it is stored (line 5).

Algorithm 4 Method *validate-variants* (as comment prefixed by //, the name of the extension point)

```
Require: Program under repair P
Require: List of program program – variants
Require: Test suite TS
1: for all pv-i ∈ program-variants do
2:     patch-i ← synthesize-patch-from-variant(P, pv-i)
3:     validation-result ← validate(TS, P, patch-i, pv-i) //EP_PV
4:     if is-valid(validation-result) //EP_FF then
5:         tsa-patches ← tsa-patches ∪ pc-i
6:     end if
7: end for
8: return tsa-patches // test suite adequate patches
```

Astor provides an extension point named EP_PV (Section 3.10) for specifying the validation process to be used by the repair approach. Built-in repair approaches over Astor use test-suite as specification of the program (Weimer et al., 2009) and as correctness

oracle. No failing test cases means the program is correct according to the specification encoded on the test suite. To validate candidate patches, Astor runs the test suite on the patched version of the buggy program.

Moreover, Astor defines an extension point named EP_FF (Section 3.11) to specify the *Fitness Function* that evaluates the patch using the output from the validation process (Algorithm 4 line 4). The result of this function is used to determine if a patch is a solution (i.e., repair the bug) or not. By default, the fitness function on Astor counts the number of failing test cases. No failing test case means the patch is a solution and is known as *test-suite adequate patch*.

2.6. Evaluation of conditions for ending navigation

An approach over Astor finishes the search of patches, i.e., stops the loop at line 7 from Algorithm 1, when any of these conditions is fulfilled (configured by the user): *a)* finding n plausible patches, *b)* iterating n times, *c)* executing during h hours (timeout).

2.7. Solution post-processing

After finishing navigating the search space, Astor provides an extension point named EP_SP (Section 3.12) for processing the patches found, if any. We envision two kinds of post-processing. First, the post-processing of each patch found aims at applying, for instance, patch minimization or code formatting. As proposed by the GenProg (Weimer et al., 2009), some changes done in a solution program variant could not be related to the bug fixing. A post-processing aims at removing such changes and keeping only those that are necessary to repair the bug. Second, the post-processing of the list of patches aims sorting patches according to a given criterion. By default, Astor lists the patches found in chronological order (first patch found, first patch listed). However, as the number of patches could be large, a repair system could order patches according to, for instance, their location, to the number of modifications each introduce, type of modification, etc.

3. Extension points provided by Astor

In this sectionn we detail the main extension points that are provided by the Astor framework for creating new repair approaches. For each extension point we give the name and description of the components already included in the framework. Table 1 summarizes all extension points.

3.1. Fault Localization (EP_FL)

3.1.1. Implemented components

- GZoltar: use of third-party library GZoltar.
- CoCoSpoon: use of third-party library CoCoSpoon.
- *Custom*: name of class that implements interface `FaultLocalizationStrategy`.

This extension point allows to specify the fault localization algorithm that Astor executes (at Algorithm 1 line 1) to obtain the buggy suspicious locations as explained in Section 2.2. The extension point takes as input the program under repair and the test suite, and produces as output a list of program locations, each one with a suspicious value. The suspicious value associated to location l goes between 0 (very low probability that l is buggy) and 1 (very high probability that l is buggy). New fault localization techniques such that PRFL presented by Zhang et al. (2017) can be implemented in this extension point.

3.2. Granularity of Modification points (EP_MPG)

3.2.1. Implemented components

- Statements: each modification point corresponds to a *statement*. Repair operators generate code at the level of *statements*.
- Expressions: each modification point corresponds to an *expression*. Repair operators generate code at the level of *expressions*.
- Logical-relational-operators: Modification points target to binary expression whose operators are logical (AND, OR) or relational (e.g., $>, ==$).
- *Custom*: name of class that implements interface `TargetElementProcessor`.

3.2.2. Description

The extension point EP_MPG allows to specify the *granularity* of code that is manipulated by a repair approach over Astor. The granularity impacts two components of Astor. First, it impacts the program representation: Astor creates modifications points only for suspicious code elements of a given granularity (Algorithm 1 line 2). Second, it impacts the repair operator space: a repair operator takes as input code of a given granularity and generates a modified version of that piece of code. For example, the approach jGenProg manipulates *statements*, i.e., the modification points refer to statements and it has 3 repair operators: add, remove and replace of statements. Differently, jMutRepair manipulates binary and unary expressions using repair operators that change binary and unary operators.

3.3. Navigation Strategy (EP_NS)

3.3.1. Implemented components

- Exhaustive: complete navigation of the search space.
- Selective: partial navigation of search space guided, by default, by random steps.
- Evolutionary: navigation of the search space using genetic algorithm.
- *Custom*: name of class that extends class `AstorCoreEngine`.

3.3.2. Description

The extension point EP_NS allows to define a strategy for navigating the search space. Algorithm 2 from section 2.5 displays a *general* navigation strategy, where most of its steps are calls to other extension points. Astor provides three navigation strategies: *exhaustive, selective* and *evolutionary*.

Exhaustive navigation. This strategy *exhaustively* navigates the search space, that is, all the candidate solutions are considered and validated. An approach that carries out an exhaustive search visits *every* modification point mp_i and applies to it *every* repair operator op_j from the repair operator space. For each combination mp_i and op_j, the approach generates zero or more candidates patches. Then, for each synthesized $patch_i$ the approach applies it into the program under repair P and then executes the validation process as explained in Section 2.5.2.

Selective navigation. The selective navigation visits a portion of the search space. This strategy is necessary when the search space is too large to be exhaustively navigated. On each step of the navigation, it uses two strategies for determining where to modify (i.e., modification points) and how (i.e., repair operators). By default, the selective navigation uses weighted random for selecting modification points, where the weight is the suspiciousness value, and uniform random for selecting operators. Those strategies can be customized using extension points EP_MPS (Section 3.4) and EP_OS (Section 3.6), respectively.

Table 1
Summary of extension points and components already implemented in Astor.

Extension point	Component	Explanation
Fault localization (EP_FL)	GZoltar	Use of third-party library GZoltar
	CoCoSpoon	Use of third-party library CoCoSpoon
Granularity modification points (EP_MPG)	Statements	Each modification point corresponds to a **statement** and repair operators generate code at the level of *statements*
	Expression	Each modification point corresponds to an expression and repair operators generate code at the level of *expressions*
	logical-relational-operators	Modification points target to binary expression whose operators are logical (AND, OR) or relational (e.g., > ==)
	if conditions	Modification points target to the expression inside *if* conditions
Navigation strategy (EP_NS)	Exhaustive	Complete navigation of the search space
	Selective	Partial navigation of search space guided, by default, by random steps
	Evolutionary	Navigation of the search space using genetic algorithm
Selection of suspicious modification points (EP_MPS)	Uniform-random	Every modification point has the same probability to be changed by an operator
	Weighted-random	The probability of changed of a modification point depends on the suspiciousness of the pointed code
	Sequential	Modification points are changes according to the suspiciousness value, in decreasing order
Operator space definition (EP_OD)	IRR-statements	Insertion, Removement and Replacement of statements
	Relational-Logical-op	Change of unary operators, and logical and relational binary operators
	Suppression	Suppression of statement, Change of if conditions by True or False value, insertion of remove statement
	R-expression	replacement of expression by another expression
Selection of operator (EP_OS)	Uniform-Random	Every repair operator has the same probability of be chosen to modify a modification point
	Weighted-Random	Selection of operator based on non-uniform probability distribution over the repair operators.
Ingredient pool definition (EP_IPD)	File	Pool with ingredients written in the same file where the patch is applied.
	Package	Pool with ingredients written in the same package where the patch is applied.
	Global	Pool with all ingredients from the application under repair.
Selection of ingredients (EP_IS)	Uniform-random	Ingredient randomly chosen from the ingredient pool
	Code-similarity-based	Ingredient chosen from similar method to that where the candidate patch is written
	Name-probability-based	Ingredient chosen based on the frequency of its variable's names
Ingredient transformation (EP_IT)	No-Transformation	Ingredients are not transformed
	Random-variable-replacement	Out-of-scope variables from an ingredients are replaced by randomly chosen in-scope variables
	Name-cluster-based	Out-of-scope variables from an ingredients are replaced by similar named in-scope variables
	Name-probability-based	Out-of-scope variables from an ingredients are replaced by in-scope variable based on the frequency of variable's names
Candidate patch Validation (EP_PV)	Test-suite	Original test-suite used for validating a candidate patch
	Augmented-test-suite	New test cases are generated for augmented the original test suite used for validation
Fitness function for evaluation (EP_FF)	Number-failing-tests	The fitness is the number of failing test cases. Lower is better. Zero means the patch is a test-suite adequate patch
Solution prioritization (EP_SP)	Chronological	Generated valid patches are printing Chronological order, according with the time they were discovered
	Less-Regression	Patches are presented according to the number of failing cases from those generated test cases, in ascending order

Evolutionary navigation. Astor framework also provides the Genetic Programming (Koza, 1992) technique for navigating the solution search space. This technique was introduced in the domain of the automatic program repair by JAFF (Arcuri, 2011) and GenProg (Weimer et al., 2009). The idea is to evolve a buggy program by applying repair operators to arrive to a modified version that does not have the bug. In Astor, it is implemented as follows: one considers an initial population of size S of program variants and one evolves them across n generations. On each generation i, Astor first creates, for each program variant pv, an offspring pvo (i.e., a new program variant) and applies, with a given probability, repair operators to one or more modification points from pvo. Then, it applies, with a given probability, the crossover operator between two program variants which involves to exchange one or more modification points. Astor finally evaluates each variant (i.e., the patch synthesized from the different operators applied) and then chooses the S variants with best fitness values (section 2.5.2) to be part of the next generation.

3.4. Selection of suspicious modification points (EP_MPS)

3.4.1. Implemented components

- Uniform-random: every modification point has the same probability to be selected and later changed by an operator.
- Weighted-random: the probability of changed of a modification point depends on the suspiciousness of the pointed code.
- Sequential: modification points are changes according to the suspiciousness value, in decreasing order.
- *Custom*: name of class that extends class `SuspiciousNavigationStrategy`.

The extension point EP_MPS allows to specify the strategy to navigate the search space of suspicious components represented by modification points. This extension point is invoked in every iteration of the navigation loop (Algorithm 1 line 7): the strategy selects the modification points where the repair algorithm will apply repair operators (Algorithm 2 line 1). Under the *uniform random* strategy, each modification point mp_x has the same probability of being selected, that is $p_u() = 1/|MP|$, where $|MP|$ is the total number of modification points considered. With the *weighted random* strategy, each modification point has a particular probability of being selected computed as follows: $p_w(mp_x) = \frac{sv_{mp_x}}{\sum_{j=1}^{|MP|} sv_{mp_j}}$, where $|MP|$ is the total number of modification points and sv_{mp_x} is the suspiciousness value of the modification point given by the fault localization algorithm.

3.5. Operator spaces definition (EP_OD)

3.5.1. Implemented components

- IRR-Statements: insertion, removement and replacement of statements.

- Relational-Logical-operators: change of unary operators, and logical and relational binary operators.
- Suppression: suppression of statement, change of if conditions by True or False value, insertion of remove statement.
- R-expression: replacement of expression by another expression.
- *Custom*: name of class that extends class `OperatorSpace`.

3.5.2. Description

After a modification point is selected, Astor selects a repair operator from the *repair operator space* to apply into that point. Astor provides the extension point EP_OD for specifying the repair operator space used by a repair approach built on Astor. The extension point is invoked at line 3 of Algorithm 1. The operators space configuration depends on the repair strategy. For example, jGenProg has 3 operators (insert, replace and remove statement) whereas Cardumen has one (replace expression).

3.6. Selection of repair operator (EP_OS)

3.6.1. Implemented components

- Uniform-random: every repair operator has the same probability of being chosen to modify a modification point.
- Weighted-random: selection of operator based on non-uniform probability distribution over the repair operators. Each repair operator has a particular probability of being chosen to modify a modification point.
- *Custom*: name of class that extends `OperatorSelectionStrategy`.

3.6.2. Description

The extension point EP_OS allows Astor's users to specify a strategy to select, given a modification point *mp*, one operator from the operator space. By default, Astor provides a strategy that applies uniform random selection and it does not depend on the selected *mp*. This strategy is used by approaches that uses selective navigation of the search space such as jGenProg and it is executed at line 4 from Algorithm 2. This extension point is useful for implementing strategies based on probabilistic models such those presented by Martinez and Monperrus (2013). In that work, several repair models are defined from different sets of bug fix commits, where each model is composed by repair operators and their associated probabilities calculated based on changes found in the commits.

3.7. Ingredient pool definition (EP_IPD)

3.7.1. Implemented components

- File: pool with ingredients written in the same file where the patch is applied.
- Package: pool with ingredients written in the same package where the patch is applied.
- Global: pool with all ingredients from the application under repair.
- *Custom*: name of class that extends the class `AstorIngredientSpace`.

3.7.2. Description

The ingredient pool contains all pieces of code that an ingredient-based repair approach can use for synthesizing a patch (section 2.4). For example, in jGenProg the ingredient pool contains all the statements from the application under repair. Then, jGenProg replaces a buggy statement by another one selected from the pool. jGenProg can also add a statement before the suspicious one.

The extension point EP_IPD allows to customize the creation of the ingredient pool. Astor provides three methods for building an ingredient pool, called "scope": file, package and global scope. When "file" scope is used, the ingredient pool contains only ingredients that are in the same file where the patch will be applied (i.e., *mp*). When the scope is "package", the ingredient pool is formed with all the code from the package that contains the modification point *mp*. Finally, when the scope is "global", the ingredient pool has all code from the program under repair. By default the original GenProg has a global ingredient scope, because it yields the biggest search space. The "file" ingredient pool is smaller than the package-one, which is itself smaller than the global one.

3.8. Selection of ingredients (EP_IS)

3.8.1. Implemented components

- Uniform-random: ingredient randomly chosen from ingredient pool.
- Code-similarity-based: ingredient chosen from similar methods to the buggy method.
- Name-probability-based: ingredient chosen based on the frequency of its variable's names.
- *Custom*: name of class that extends class IngredientSearchStrategy.

3.8.2. Description

The extension point EP_IS allows to specify the strategy that an ingredient-based repair approach from Astor uses for selecting an ingredient from the ingredient pool. Between the implementations of this point provided by Astor, One, used by default by jGenProg, executes uniform random selection for selecting an ingredient from a pool built given a scope (see Section 3.7). Another, defined for DeepRepair approach, prioritizes ingredients that come from methods which are similar to the buggy method.

3.9. Ingredient transformation (EP_IT)

3.9.1. Implemented components

- No-transformation: ingredients are not transformed.
- Random-variable-replacement: out-of-scope variables from an ingredients are replaced by randomly chosen in-scope variables.
- Name-cluster-based: out-of-scope variables from an ingredients are replaced by similar named in-scope variables.
- Name-probability-based: out-of-scope variables from an ingredients are replaced by in-scope variable based on the frequency of variable's names.
- *Custom*: name of class that extends class `IngredientTransformationStrategy`.

3.9.2. Description

The extension point EP_IT allows to specify the strategy used for transforming ingredients selected from the pool. Astor provides four implementations of this extension point. For instance, the strategy defined for DeepRepair approach replaces each out-of-scope variable form the ingredient by one variable in the scope of the modification points. The selection of that variable is based on a cluster of variable names, which each cluster variable having semantically related names (White et al., 2017). Cardumen uses a probabilistic model for selecting the most frequent variables names to be used in the patch. On the contrary, jGenProg, as also the original GenProg, does not transform any ingredient.

3.10. Candidate Patch Validation (EP_PV)

3.10.1. Implemented components

- Test-suite: original test-suite used for validating a candidate patch.

- Augmented-test-suite: new test cases are generated for augmented the original test suite used for validation.
- *Custom*: name of class that extends class `ProgramVariantValidator`.

3.10.2. Description

The extension point EP_PV executes the validation process of a patch (Section 2.5.2). Astor framework provides to test-suite based repair approaches a validation process that runs the test-suite on the patched program. The validation is executed in Algorithm 2 line 3.

Another strategy implemented in Astor was called MinImpact (Yu et al., 2017), proposed to alleviate the problem of patch overfitting (Smith et al., 2015). MinImpact uses additional automatically generated test cases to further check the correctness of a list of generated test-suite adequate patches and returns the one with the highest probability of being correct. MinImpact implements the extension point EP_PV by generating new test cases (i.e., inputs and outputs) over the buggy suspicious files, using Evosuite (Fraser and Arcuri, 2011) as test-suite generation tool. Once generated the new test cases, MinImpact executes them over the patched version. The intuition is that the more additional test cases fail on a tentatively patched program, the more likely the corresponding patch is an overfitting patch. MinImpact then sorts the generated patches by prioritizing those with less failures over the new tests.

Moreover, this extension point can be used to measure other functional and not functional properties beyond the verification of the program correctness. For example, instead of focusing on automated software repair, an approach built over Astor could target on minimizing the energy computation. For that, that approach would extend this extension point for measuring the consumption of a program variant.

3.11. Fitness Function for evaluating candidate (EP_FF)

3.11.1. Implemented components

- Number-failing-tests: the fitness is the number of failing test cases. Lower is better. Zero means the patch is a test-suite adequate patch.
- *Custom*: name of class that implements `FitnessFunction`.

3.11.2. Description

The extension point EP_FF allows to specify the *fitness function*, which consumes the output from the validation process of a program variant *pv* and assigns to *pv* its fitness value. Astor provides an implementation of this extension point which considers as fitness value the number of failing test cases (low is better).

On evolutionary approaches (Section 3.3.2) such as jGenProg, this fitness function guides the evolution of a population of program variants throughout a number of generations. In a given generation t, those variant with better fitness will be part of the population at generation $t + 1$. On the contrary, on selective or exhaustive approaches, the fitness function is only used to determined if a patched program is solution or not.

3.12. Solution prioritization (EP_SP)

3.12.1. Implemented components

- Chronological: generated valid patches are printing chronological order, according with the time they were discovered.
- Less-regression: patches are presented according to the number of failing cases from those generated test cases, in ascending order.
- *Custom*: name of class that implements `SolutionVariantSortCriterion`

3.12.2. Description

The extension point EP_SP allows to specify a method for sorting the discovered valid patches. By default, approaches over Astor present patches sorted by time of discovery in the search space. Astor proposes an implementation of this point named *Less-regression*. The strategy, defined by MinImpact (Yu et al., 2017), sorts the *original test-suite* adequate patches with the goal of minimizing the introduction of regression faults, i.e., the approach prioritizes the patches with *less failing test cases* from those tests automatically generated during the validation process.

4. Repair Approaches implemented in Astor

In this section we present a brief description of repair approaches built over Astor framework and publicly available at Github platform. Those approaches were built combining different components implemented for the extension points presented in Section 3. Table 2 displays the components that form each built-in repair approach from Astor. The approaches are presented in the order they were introduced into Astor framework.

4.1. jGenprog

jGenProg is an implementation of GenProg (Weimer et al., 2009) built over Astor framework. The approach belongs to the family of ingredient-based repair approaches (Section 2.4) and it has 3 repairs operators: insert, replace and remove statements. For the two first mentioned operators, jGenProg uses statements written somewhere in the application under repair for synthesizing patches that insert or replace statement. jGenProg can navigate the search space (Section 3.3.2) in two ways: *a)* using evolutionary search, as the original GenProg does; or *b)* using selective, as RSRepair (Qi et al., 2014) does.

4.2. jKali

The technique Kali was presented by Qi et al. (2015b) for evaluating the incompleteness test suites used by repair approaches for validating candidate patches. The intuition of the authors was that removing code from a buggy application was sufficient to pass all test from incomplete test suite. Consequently, the generated patches overfit the incomplete test suite and are not valid of inputs not included on it. jKali is an implementation of Kali built over Astor framework which removes code and skips the execution of code by adding `return` statements and turning True/False expressions from `if` conditions. As Kali, jKali does an exhaustive navigation of the search space (Section 3.3.2).

4.3. jMutRepair

Mutation-based repair system was introduced by Debroy and Wong (2010) to repair bugs using mutation operators proposed by mutation testing approaches (DeMillo et al., 1978; Hamlet, 1977). We implemented that system over Astor, named jMutRepair, with has as repair operators the mutation of relational (e.g.,$==$, $>$) and logical operators (e.g., AND, OR). jMutRepair does an exhaustive or selective navigation of the search space (Section 3.3.2).

4.4. DeepRepair

DeepRepair (White et al., 2017) is an ingredient-based approach built over jGenProg that applies *deep learning* based techniques during the patch synthesis process. In particular, DeepRepair proposes new strategies for: *a)* selecting ingredients (section 3.8) based on similarity of methods and classes: ingredients are taken from the most similar methods or classes to those that contains

Table 2
Main Extension points and decision adopted by each approach. Each approach and extension point includes a reference to the section that explain it.

Name Extension Point	jGenProg 4.1	jKali 4.2	jMutRepair 4.3	DeepRepair 4.4	Cardumen 4.5	TIBRA 4.6
3.1 Fault localization (EP_FL)	GZoltar	-	GZoltar	GZoltar	GZoltar	GZoltar
3.2 Granularity of modification points (EP_MPG)	Statement	Statements + *if*	Relational-Logical-operators	Statement	Expression	Statement
3.3 Navigation strategy (EP_NS)	Selective or Evolutionary	Exhaustive	Exhaustive or Selective	Selective or Evolutionary	Selective	Selective
3.4 Selection susp. points (EP_MPS)	Weighted-random	Sequence	Weighted-random	Weighted-random	Weighted-random	Weighted-random
3.5 Operator space definition (EP_OD)	IRR-statements	Suppression	Relational-Logical-operators	Suppression	R-expression	Weighted-random
3.6 Selection operator (EP_OS)	Random	Sequential	Random	Random	-	Random
3.7 Ingredient pool definition (EP_IPD)	Package	-	-	Package	Global	Package
3.8 Selection ingredients (EP_IS)	Uniform-random	-	-	Code-similarity-based	Uniform-random	Uniform-random
3.9 Ingredient transformation (EP_IT)	No-transf.	-	-	Name-cluster-based	Name-probability-based	Random-variable-replacement
3.10 Candidate patch validation (EP_PV)	Test-suite	Test-suite	Test-suite	Test-suite	Test-suite	Test-suite
3.11 Fitness function (EP_FF)	#failing-tests	#failing-tests	#failing-tests	#failing-tests	#failing-tests	#failing-tests
3.12 Solution prioritization (EP_SP)	Chronol.	Chronol.	Chronol.	Chronol.	Chronol.	Chronol.

the bug; *b)* transforming ingredients (Section 3.9) based on semantic of variables names: out-of-scope variables from an ingredients are replaced by in-scope variables with semantically-related names.

4.5. Cardumen

Martinez and Monperrus (2018) is a repair system that targets fine-grained code elements: *expressions*. It synthesizes repairs from templates mined from the applications under repair. Then, it creates concrete patches from those templates by using a probabilistic model of variable names: it replaces template placeholders by variables with the most frequent names at the buggy location. Cardumen explores the search space using the selective strategy (Section 3.3.2).

4.6. TIBRA

TIBRA (Transformed Ingredient-Based Repair Approach), is an extension of jGenProg which, as difference with the original GenProg who discards ingredients with out-of-scope variables, it applies transformation into the ingredients. The approach adapts an ingredient i.e., statement taken somewhere, by replacing all variables out-of-scope from it by in-scope variables randomly chosen from the buggy location.

5. Evaluation

In this section we present an evaluation of repair approaches built over Astor. We first study the capacity of approaches presented in section 4 to repair real bugs. Then, we focus on the *ingredient pool*, the component of GenProg that stores the source code used for synthesizing candidate patches. There, we compare different implementations of the extension points from Section 3 related to the creation of the ingredient pool and to the transformation of ingredients.

5.1. Research questions

The research questions that guide the evaluation of Astor are:

1) Repairability:
   - RQ 1.1 - How many bugs from Defects4J are repaired using the repair approaches built over Astor?
   - RQ 1.2: What are the bugs uniquely repaired by approaches from Astor?
   - RQ 1.3 - Which code granularity repairs more bugs?
2) Focus on ingredient-based repair:
   - RQ 2.1 -To what extent does a reduced ingredient space impact repairability?
   - RQ 2.2 - To what extent does the ingredient transformation strategy impact repairability?

5.2. Protocol

We have used repair approaches from Astor for a large evaluation consisting in searching for patches for 357 bugs from the Defects4J benchmarks (Just et al., 2014), those from 5 projects: Apache Commons Math, Apache Commons Lang, JFreeChart, Closure and Joda Time. A patch is said to be *test-adequate* if it passes all tests, including the failing one. As shown by previous work (Qi et al., 2015a), a patch may be test-adequate yet incorrect, when it only works on the inputs from the test suite and does not generalize. Those patches are known as *overfitting* patches (Smith et al., 2015; Le et al., 2018; Yu et al., 2018). This paper does not aim at studying the correctness of the test-suite adequate patches found by a repair approach as done in, for instance, in Martinez et al. (2016). In the rest of the paper, *"to repair a bug"* means to find a test-suite adequate patch.

The main experimental procedure is composed of the following steps. First, we selected the repair approaches to study according to the addressed research question. Then, we created scripts for launching repairs attempts for each approach on each bug from Defects4J. A repair attempt consists on the execution of a repair approach executed with a timeout of 3 hours and a concrete value of random seed. In case of exhaustive approach such as jKali, the seed is not used due it does not have any stochastic sub-component. For each configuration, we run at least 3 repair attempts (i.e., 3 different seeds). Finally, we collected the results from each repair attempt, grouped the results (i.e., patches and statistics) according to the configuration, and compared the results.

5.3. Evaluation of Repairability

In this section we focus on the ability of repair approaches over Astor introduced in Section 4 to repair bugs from Defects4J and we compare their repairability against other repairs systems.

5.3.1. RQ 1.1 - How many bugs from Defects4J are repaired using the repair approaches built over Astor?

Table 3 displays the bugs from Defects4J repaired by approaches built over Astor framework. In total, 98 bugs out of 357 (27.4%)

Table 3
Bugs from dataset Defects4J repaired by approaches built over Astor. In total, 98 bugs from 5 Java projects were repaired. R means 'bug with at least one test-suite adequate patch'.

Project	Bug Id	jGenProg	jKali	jMutRepair	DeepRepair	Cardumen	TIBRA	# approaches	Project	Bug Id	jGenProg	jKali	jMutRepair	DeepRepair	Cardumen	TIBRA	# approaches
Chart	1	R	R	R	R	R	R	6	Math	24				R			1
	3	R			R	R	R	4		28	R	R	R	R	R	R	6
	4					R		1		30					R		1
	5	R	R		R	R	R	5		32	R	R		R	R	R	5
	6					R	R	2		33					R		1
	7	R		R	R	R	R	5		39	R						1
	9				R	R		2		40	R	R	R	R	R		5
	11					R	R	2		41					R		1
	12	R			R	R	R	4		44	R					R	2
	13	R	R		R	R		4		46					R		1
	14				R			1		49	R	R		R	R	R	5
	15	R	R		R	R		4		50	R	R	R	R	R	R	6
	17					R	R	2		53	R			R		R	3
	18				R			1		56	R			R			2
	19	R						1		57			R	R	R		3
	23						R	1		58			R	R	R		3
	24					R	R	2		60	R			R	R		3
	25	R	R	R	R	R		5		62					R		1
	26	R	R	R	R	R	R	6		63				R	R	R	3
Closure	7		R			R		2		64	R						1
	10		R	R		R		2		69					R		1
	12					R		1		70	R			R	R		3
	13		R			R		2		71	R			R			2
	21	R	R	R		R		3		72						R	1
	22	R	R	R		R		4		73	R			R	R	R	4
	33					R		1		74	R			R	R		3
	38			R				1		77				R			1
	40					R		1		78	R	R		R	R	R	5
	45		R			R		2		79					R	R	2
	46	R	R			R		3		80	R	R	R	R	R	R	6
	49	R						1		81	R	R	R	R	R	R	6
	55					R		1		82	R	R	R	R	R	R	6
	133					R		1		84	R	R	R	R	R		5
Lang	7	R			R	R		3		85	R	R	R	R	R	R	6
	10	R			R	R		3		88			R		R		2
	14					R		1		95	R	R			R	R	4
	20				R			1		97					R	R	2
	22	R			R	R		3		98				R			1
	24	R			R	R		3		101					R		1
	27	R		R	R	R		4		104			R		R		2
	38				R			1		105					R		1
	39	R			R	R		3	Time	4	R				R	R	3
Math	2	R	R	R	R	R	R	6		7					R		1
	5	R			R	R	R	4		9					R		1
	6				R	R		2		11	R	R	R		R	R	5
	7	R						1		17					R		1
	8	R	R		R	R	R	5		18					R		1
	18				R	R		2		20						R	1
	20	R	R		R	R	R	5									
	22				R			1	Σ	98	49	29	23	51	77	35	

are repaired by at least one repair approach. Six approaches were executed: jGenProg (49 bugs repaired), jKali (29), jMutRepair (23), DeepRepair (51), Cardumen (77), and TIBRA (35).

We observe that there are 9 bugs such as Chart-1 and Math-2 that all evaluated repair approaches found at least one test-suite adequate patches. Contrary, 35 bugs were repaired by only one approach. 19 of them, such as Math-101, are repaired by Cardumen, 9 only by DeepRepair (e.g., Math-22), 3 by jGenProg (e.g., Chart-19), 3 by TIBRA (e.g., Chart-23), and one for jMutRepair (Closure-38).

Response to RQ 1.1: The repair approaches built over Astor find test-adequate patches for **98** real bugs out of 357 bugs from Defects4J. The best approach is Cardumen: it finds a test-suite adequate patch for 77 bugs.

Compared with other repair system evaluated over Defects4J, approaches from Astor repair more bugs (98 bugs repaired) from Defects4J than: ssFix (Xin and Reiss, 2017b) (60 bugs repaired), ARJA (Yuan and Banzhaf, 2017) (59 bugs), ELIXIR (Saha et al., 2017) (40 bugs), GP-FS (Wen et al., 2017) (37 bugs), JAID (Chen et al., 2017) (31 bugs), ACS (Xiong et al., 2017) (18 bugs), HDRepair (Le et al., 2016) (15 bugs from Xin and Reiss (2017b)). In particular, Cardumen repairs more that all those approaches: it found test-suite adequate patches for 77 bugs. On the contrary, Nopol (Xuan et al., 2017) (103 bugs repaired (Durieux et al., 2017b)) repairs more than Astor framework.

5.3.2. RQ 1.2: What are the bugs uniquely repaired by approaches from Astor?

We consider automated program repair approaches from the literature for which: *1)* the evaluation was done over the dataset Defects4J; *2)* the identifiers of the repaired bugs from Defect4J are given on the respective paper or included in the appendix.

We found 10 repair systems that fulfill both criteria: Nopol (Xuan et al., 2017; Durieux et al., 2017b), jGenProg (Martinez et al., 2016), DynaMoth (Durieux and Monperrus, 2016), HDRepair (Le et al., 2016), DeepRepair (White et al., 2017), ACS (Xiong et al., 2017), GP-FS (Wen et al., 2017), JAID (Chen et al., 2017), ssFix (Xin and Reiss, 2017b) and ARJA (Yuan and Banzhaf, 2017). In the case of HDRepair, as neither the identifiers of the repaired bugs nor the actual patches were reported by Le et al. (2016), we considered the results reported by Xin and Reiss (2017b) (ssFix's authors) who executed the approach. We discarded the Java systems JFix (Le et al., 2017a), S3 (Le et al., 2017b), Genesis (Long et al., 2017) (evaluated over different bug datasets) and ELIXIR (Saha et al., 2017) (repaired ids from Defect4J not publicly available).

Approaches built on Astor framework found test-suite adequate patches for **11** new bugs of Defects4J, for which no other system ever has managed to find a single one. Those uniquely repaired bugs are: 5 from Math (ids: 62, 64, 72, 77, 101), 2 from Time (9, 20), 2 from Chart (11, 23), and 2 from Closure (13, 46).

Response to RQ 1.2: The repair approaches built over Astor find new unique test-adequate patches. Astor repair **11** bugs which have never been repaired previously by any repair system.

In the remain of the evaluation section, we evaluates different section implementation for tree extension points from 3.

5.3.3. RQ 1.3 - Which code granularity repairs more bugs?

We compared the repairability of two approaches that use different implementations of the extension point EP_MPG for manipulating different granularity of code source. The level of code granularity impacts on the size of the search space and thus in the ease to find a patch. On one hand, Cardumen approach is able to synthesize a fine-grained patches by modifying, for example, an expression inside a statement with other expressions inside. On the other hand, the code modifications done by jGenProg (or by any approach that extends it such as DeepRepair or TIBRA) to find a patch are coarse-level: as it works at the level of *statements*, an entire statement is inserted, deleted or replaced.

Table 3 shows that Cardumen (expression level) and jGenProg's family approaches (statement level) repaired 77 and 72 bugs, respectively, and 52 of them were repaired by both approaches. Even the difference of repairability is not statistically significant enough (Mann-Whitney test shows a p-value of 0.649), the experiment shows that there is a considerable portion of bugs that can be repaired only in a given granularity: 25 bugs are repaired by Cardumen but not by jGenProgs family, and 20 bugs are repaired by only this latter family of approaches.

Response to RQ 1.3: The extension point EP_MPG has an impact on repair. We compared the extensions *statements* and *expression* implemented in jGenProg and Cardumen, respectively. By applying operators at the level of statements and expressions, **52** patches (72.2% and 67.5%, resp.) are repaired by both jGenProg and Cardumen, respectively. The remaining bugs (20 and 25, resp.) are only repaired using a specific granularity.

The implication is that, for those bugs repaired by both approaches, there are patches that: *a)* produce similar behaviours w.r.t the test-suite (i.e., passing all tests), and *b)* the changed codes have different granularities.

For example, both jGenProg and Cardumen synthesize the following patch for bug Math-70:

```
72 -            return solve(min, max);
72 +            return solve(f, min, max);
       }
```

Listing 2. Patch for Math-70 at class BisectionSolver.java.

jGenProg synthesizes that patch by applying the replace operator to the modification point that references to the `return` *statement* at line 72 of class BisectionSolver. The replacement `return` statement (i.e., the ingredient) is taken from line 59 from the same buggy class. Meanwhile, Cardumen arrives to the same patch by replacing a modification point that references to the *expression* corresponding to the method invocation `solve` inside the `return` statement at line 72. The replacement synthesized from a template:

`solve(_UnivariateRealFunction_0, _double_1, _double_2)` mined from the same class BisectionSolver and instantiated using variables in scope at line 72.

One of the 25 bugs repaired by Cardumen but not by jGenProg is Math-101 The patch proposed modifies the expression related to a variable initialization (endIndex) at line 376 from `startIndex + n` to `source.length()` on class ComplexFormat.

```
376 -    int endIndex = startIndex + n;
376 +    int endIndex = source.length();
```

Listing 3. Patch for Math-101 by Cardumen.

Cardumen is able to synthesize the patch by instantiating the template `_String_0.length()` mined from the application under repair. Approaches working at a different (and coarse) granularity are not capable to synthesize that patch: in the case of jGenProg, the statement `int endIndex = source.length();` (the ingredient for the fix) does not exist anywhere in the application under repair.

Astor framework provides to developers of approaches the flexibility to manipulate specific code elements at a given granularity level by implementing the extension point EP_MPG. For instance, jMutRepair implements EP_MPG to manipulate relational and logical binary operator. This allows to target to specific defect classes and to reduce the search space size. That is the case for bug Closure-38, which is only repaired by jMutRepair.

```
245 - if (x < 0) && (prev == '-') {
245 + if (x <= 0) && (prev == '-') {
```

Listing 4. Patch for Closure-38 by jMutRepair at class CodeConsumer.

5.4. Design of ingredient-based repair approaches

In this section we study and compare two different implementations for the extension points related to ingredient-based repair approaches EP_IPD and EP_IT. The goal of the next experiments is to study whether improvements introduced via those extension points impact on the repairability with respect to the vanilla jGenProg.

5.4.1. RQ 2.1 -To what extent does a reduced ingredient space impact repairability?

We evaluate the extension point EP_IPD by using different strategies for building an ingredient pool (Section 3.7). The experiment's goal is to know whether using a reduced ingredient pool, such as File and Package pools preserves the repair capability of a the vanilla GenProg approach (which uses Global scope as default).

To study the impact of using a reduced ingredient pool, we executed jGenProg on Defects4J using the baseline ingredient pool used by the original GenProg (Weimer et al., 2009) (i.e, Global pool) and the optimized modes (File and Package pool), based on the empirical evidences of ingredient's locations (Martinez et al., 2014; Barr et al., 2014). For each bug from Defects4J and for each pool type (File, Package and Global) we executed 3 repairs attempts with a timeout of 3 hours, using on each attempt a different random seed value.

In this experiment, jGenProg repaired, in total, 39 bugs using ingredient based repair operators.[2] The numbers of repaired bugs are different according with the ingredient pool used by jGenProg: it repaired 33, 28 and 14 bugs using File, Package and Global pool, respectively. We applied the Wilcoxon rank sum test (aka Mann-Whitney test) to verify that the difference of repairability between jGenProg using a reduced space and jGenProg using Global are statically significant, obtaining p-values of 1.296e-05 (File vs Global) and 0.001613 (Package vs Global).

Moreover, we found that *all* the bugs repaired using Global scope were also repaired by jGenProg using either File or Package scopes. This means that a reduced ingredient space still continue having, at least, one ingredient that is used to synthesize a test-suite adequate patch.

The reduction of the ingredient space produces another advantage: it allows jGenProg to find faster the first test-suite adequate patch. For the 12 bugs that were repaired jGenProg using both File and Global scopes, 10 of them were repaired faster using File scope (83%), saving on average 22.8 minutes. Similarly, for the 14 bugs that were repaired using both Package and Global scopes, 12 of them were repaired faster using package. This result validates the fact that locality-aware repair speeds-up repair (Barr et al., 2014; Martinez et al., 2014).

> **Response to RQ 2.1:** The extension point EP_IPD impacts the numbers of repaired bugs and on the repair time. We compared jGenProg using the File, Package and Global (default by GenProg) scopes for building the ingredient pool. For our repair attempts bounded by a 3-hours maximum budget, the File and Package ingredients pools found more test-suite adequate patches and faster (reduction of 22 minutes on average) than the baseline (Global).

5.4.2. RQ 2.2 - To what extent does the ingredient transformation strategy impact repairability?

By default, the original GenProg does not apply any transformation of ingredients once they are selected from the ingredient search space. As we presented in Section 3.9 Astor provides an extension point EP_IT for plugging an ingredient transformation strategy. We now present the evaluation of two different implementations of EP_IT, both included in Astor.

First, we evaluated the approach TIBRA (Section 4.6), which replaces out-of-scope variables from an ingredient, and we compared the results against those from the original jGenProg (which does not transformation ingredients selected from the search space). TIBRA repaired 11 bugs (e.g., Math-63 in Table 3) that jGenProg did not repair, showing that the transformation of ingredient allows to find patches for unrepaired bugs. The Wilcoxon signed-rank test shows that the difference between the repairability from vanilla jGenProg and TIBRA is statistically significant, with a p-value of 1.247e-09.

A second implementation of the extension point EP_IT is a cluster-based ingredient transformation strategy proposed by DeepRepair (White et al., 2017) (which is built over Astor). We carried out a second experiment that compared jGenProg using this strategy (named DeepRepair RE in White et al. (2017)) with the original jGenProg. The results showed that there is not a statistical differences between the repairability of bugs from Defects4j.[3] The transformation of ingredients using the cluster-based strategy allows to repair only 4 bugs that jGenProg could not (e.g., Math-98 in Table 3). However, we found that there are notable differences between DeepRepair and jGenProg patches: 53%, 3%, and 53% of DeepRepair's patches for Chart, Lang, and Math, respectively, are not found by jGenProg.

Finally, we compared TIBRA and DeepRepair. TIBRA repairs in total 35 bugs, of them 21 are also repaired by DeepRepair. This means that 28 and 14 bugs are only repaired by DeepRepair (RE) and TIBRA, respectively. The Wilcoxon signed-rank test shows that the difference between the repairability from DeepRepair (RE) and TIBRA is statistically significant, with a p-value of 0.02739. This last experiment shows the benefits of using a customized strategy based on cluster of variables names over a strategy based on random selection of variables.

[2] In this experiment we did not consider all bugs repaired using the remove operator.

[3] Wilcoxon test shows a p-value = 0.3626. Consequently, in White et al. (2017) we fail to reject the Null Hypothesis that states that the DeepRepair's ingredient transformation strategy RE generates the same number of test-adequate patches as jGenProg.

Response to RQ 2.2: The extension point EP_IT impacts the repairability. We compared three extensions: *no-transformation, cluster-based*, and *random-variable-replacement* implemented in jGenProg, DeerRepair and TIBRA, respectively. DeepRepair and TIBRA discover new test-suite adequate patches that cannot by synthesized by jGenProg for 4 and 11 bugs, respectively.

6. Related Work

6.1. Program Repair Frameworks

To our knowledge, Astor is the first and unique framework on Java that implements a repair workflow from generate-and-validate repair approaches and provides twelve extension points for which the program repair researcher can either choose an existing component (i.e., taking one already implemented design decision in the design space), or can implement a new technique.

6.2. Works that extend approaches from Astor

In this section we present the repair approaches and extensions from the bibliography that were built over the Astor framework. Tanikado et al. (2017) extended jGenProg provided by Astor framework for introducing two novel strategies. One, named *similarity-order*, which extends extension point EP_IS, chooses ingredients according to code fragment similarities. The second one, named *freshness-order*, which extends the modification point EP_MPS, consists on selecting, with a certain priority, modification points whose statements were more recently updated. Wen et al. (2017) presented a systematic empirical study that explores the influence of fault space on search-based repair techniques. For that experiment, the author created the approach GP-FS, an extension of jGenProg, which receives as input a faulty space. In their experiment, the authors generated several fault spaces with different accuracy, finding that GP-FS is capable of fixing more bugs correctly when fault spaces with high accuracy are fed. White et al. (2017) presented DeepRepair, an extension of jGenProg, which navigates the search space guided by method and class similarity measures inferred with deep unsupervised learning. DeepRepair was incorporated to Astor framework as built-in approach.

6.3. Works that execute built-in approaches from Astor

Works from the literature executed repair approaches from Astor framework during the evaluation of their approaches. For example, Yuefei presents and study (Liu, 2017) for understanding and generating patches for bugs introduced by third-party library upgrades. The author run jGenProg from Astor to repair the 6 bugs, finding correctly 2 patches for bugs, and a test-suite adequate but yet incorrect patch for another bug. The approach ssFix (Xin and Reiss, 2017b) performs syntactic code search to find existing code from a code database (composed by the application under repair and external applications) that is syntax-related to the context of a bug statement. In their evaluation, the authors executed two approaches from Astor, jGenProg and jKali, using the same machines and configuration that used for executing ssFix.

6.4. Works that compare repairability against that one from built-in approaches from Astor

We have previously executed jGenProg and jKali over bugs from Defects4J (Just et al., 2014) and analyzed the correctness of the generated patches (Martinez et al., 2016). Note that, the number of repaired bugs we reported in that experiment, executed in 2016, are lower that the results we present in this paper in Section 5. The main reason is we have applied several improvements and bugfixings over Astor framework since that experiment.

Other works have used the mentioned evaluation of jGenProg and jKali presented in Martinez et al. (2016) for measuring the improvement introduced by their new repair approaches. For example, Le et al. presented a new repair approach named HDRepair (Le et al., 2016) which leverages on the development history to effectively guide and drive a program repair process. The approach first mines bug fix patterns from the history of many projects and the then employ existing mutation operators to generate fix candidates for a given buggy program. The approach ACS (Automated Condition Synthesis) (Xiong et al., 2017), targets to insert or modify an âifâ condition to repair defects by combining three heuristic ranking techniques that exploit 1) the structure of the buggy program, 2) the document of the buggy program (i.e., Javadoc comments embedded in the source code), and 3) the conditional expressions in existing projects. Yuan and Banzhaf (2017) present ARJA, a genetic-programming based repair approach for automated repair of Java programs. ARJA introduces a test filtering procedure that can speed up the fitness evaluation and three types of rules that can be applied to avoid unnecessary manipulations of the code. ARJA also considers the different representation of ingredient pool introduced by Astor framework (Martinez and Monperrus, 2016). In addition to the evaluation of Defects4J, the authors evaluated the capacity of repair real multi-location bugs over another dataset built by themselves. Saha et al. presented Elixir (Saha et al., 2017) a repair technique which has a fixed set of parameterized program transformation schemas used for synthesized candidate patches. JAID by Chen et al. (2017) is a state-based dynamic program analyses which synthesizes patches based on schemas (5 in total). Each schema trigger a fix action when a suspicious state in the system is reached during a computation. JAID has 4 types of fix actions, such as modify the state directly by assignment, and affect the state that is used in an expression.

6.5. Works that analyze patches from built-in approaches from Astor

Other works have analyzed the publicly available patches of jGenProg and jKali from our previous evaluation of repair approaches over Defects4J dataset (Martinez et al., 2016). Motwani et al. (2017) analyzed the characteristics of the defects that repair approaches (including jGenProg and jKali) can repair. They found that automated repair techniques are less likely to produce patches for defects that required developers to write a lot of code or edit many files. They found that the approaches that target Java code, such as those from Astor, are more likely to produce patches for high-priority defects than the techniques which target C code. Yokoyama et al. (2017) extracted characteristics of defects from defect reports such as priority and evaluated the performance of repairs approaches against 138 defects in open source Java project included in Defects4J. They found that jGenProg is able to find patch for many high-priority defects (1 Blocker, 2 Critical, and 11 Major). Liu et al. (2017) presented a approach that heuristically determines the correctness of the generated patches, by exploiting the behavior similarity of test case executions. The approach is capable of automatically detecting as incorrect the 47.1% and 52.9% of patches from jGenprog and jKali, respectively. Jiang and Xiong (2017) analyzed the Defects4J dataset for finding bugs with weak test cases. They results shows that 42 (84.0%) of the 50 defects could be fixed with weak test suites, indicating that, beyond the current techniques have a lot of rooms for improvement, weak test suites may not be the key limiting factor for current techniques.

6.6. Other test-suite based repair approaches

During the last decade, other approaches target other programming languages (such as C) or we evaluated over other datasets rather than Defects4J were presented. Arcuri (2011) applies coevolutionary computation to automatically generate bug fixes for Java program. GenProg (Weimer et al., 2009; Goues et al., 2012b), one of the earliest generate-and-validate techniques, uses genetic programming to search the repair space and generates patches created from existing code from elsewhere in the same program. It has three repair operators: add, replace or remove statements. Other approaches have extended GenProg: for example, AE (Weimer et al., 2013) employs a novel deterministic search strategy and uses program equivalence relation to reduce the patch search space. The original implementation (Weimer et al., 2009) targets C code and was evaluated against dataset with C bugs such as ManyBugs and IntroClass (Goues et al., 2015). Astor provides a Java version of GenProg called jGenProg which also employs genetic programming for navigating the search space. RSRepair (Qi et al., 2013a) has the same search space as GenProg but uses random search instead, and the empirical evaluation shows that random search can be as effective as genetic programming. Astor is able to execute a Java version of RSRepair by choosing random strategies for the selection of modification points (extension point EP_MPS) and operators (extension point EP_OS). Debroy and Wong (2010) propose a mutation-based repair method inspired from mutation testing. This work combines fault localization with program mutation to exhaustively explore a space of possible patches. Astor includes a Java version of this approach called jMutRepair. Kali (Qi et al., 2015a) has recently been proposed to examine the fixability power of simple actions, such as statement removal. As GenProg, Kali targets C code. Astor proposes a Java version of Kali, which includes all transformations proposed by Kali.

Other approaches have proposed new set of repair operators. For instance, PAR (Kim et al., 2013), which shares the same search strategy with GenProg, uses patch templates derived from human-written patches to construct the search space. The PAR tool used the original evaluation is not publicly available. However, it is possible to implement PAR over the Astor framework by implementing the repair operator based on those templates using the extension point EP_OD. The approach SPR (Long and Rinard, 2015) uses a set of predefined transformation schemas to construct the search space, and patches are generated by instantiating the schemas with condition synthesis techniques. SPR is publicly available but targets C programs. An extension of SPR, Prophet (Long and Rinard, 2016) applies probabilistic models of correct code learned from successful human patches to prioritize candidate patches so that the correct patches could have higher rankings.

There are approaches that leverage on human written bug fixes. For example, Genesis (Long et al., 2017) automatically infers code transforms for automatic patch generation. The code transformation used Genesis are automatically infer from previous successful patches. The approach first mines bug fix patterns from the history of many projects and the then employ existing mutation operators to generate fix candidates for a given buggy program. Both approaches need as input, in addition to the buggy program and its test suite, a set of bug fixes. Two approaches leveraged on semantics-based examples. SearchRepair (Ke et al., 2015) uses a large database of human-written code fragments encore as satisfiability modulo theories (SMT) constraints on their input-output behavior for synthesizing candidates repairs. S3 (Syntax- and Semantic-Guided Repair Synthesis) by Le et al. (2017b), a repair synthesis engine that leverages programming-by-examples methodology to synthesize repairs.

Other approaches belong to the family of *synthesis-based repair approaches*. For example, SemFix (Duong et al., 2013) is a constraint based repair approach for C. This approach provides patches for assignments and conditions by combining symbolic execution and code synthesis. Nopol (Xuan et al., 2017) is also a constraint based method, which focuses on fixing bugs in if conditions and missing preconditions, as Astor, it is implemented for Java and publicly available. DynaMoth (Durieux and Monperrus, 2016) is based on Nopol, but replaces the SMT-based synthesis component of Nopol by a new synthesizer, based on dynamic exploration, that is able to generate richer patches than Nopol e.g., patches on *If* conditions with method invocations inside their condition. DirectFix (Mechtaev et al., 2015) achieves the simplicity of patch generation with a Maximum Satisfiability (MaxSAT) solver to find the most concise patches. Angelix (Mechtaev et al., 2016) uses a lightweight repair constraint representation called “angelic forestâ to increase the scalability of DirectFix.

6.7. Studies analyzing generated patches

Recent studies have analyzed the patches generated by repair approaches from the literature. The results of those studies show that generated patches may just overfit the available test cases, meaning that they will break untested but desired functionality. For example, Qi et al. (2015a) find, using Kali system, that the vast majority of patches produced by GenProg, RSRepair, and AE avoid bugs simply by functionality deletion. A subsequent study by Smith et al. (2015) further confirms that the patches generated by GenProg and RSRepair fail to generalize.

Due to the problematic of test overfitting, recent works by Xiong et al. (2018), and Yu et al. (2017) propose to extend existing automated repair approach such as Nopol, ACS and jGenProg. Those extended approaches generate new test inputs to enhance the test suites and use their behavior similarity to determine patch correctness. For example, Liu reported (Xiong et al., 2018) that their approach, based on patch and test similarity analysis, successfully prevented 56.3% of the incorrect patches to be generated, without blocking any correct patches. Yang et al. presented a framework named Opad (Overfitted PAtch Detection) (Yang et al., 2017) to detect overfilled patches by enhancing existing test cases using fuzz testing and employing two new test oracles. Opad filters out 75.2% (321/427) overfitted patches generated by GenProg/AE, Kali, and SPR.

7. Threat to Validity

Internal validity. A threat to the validity of our results relates to whether our implementation of existing repair approaches are faithful to the original C implementation. We have developed jGenProg, jKali and jMutRepair based on our deep analysis of the algorithms of GenProg, Kali and Mutation Repair, respectively, presented in different publications (Weimer et al., 2009; Qi et al., 2015a; Debroy and Wong, 2010). In the case of GenProg, we clearly see GenProg as two separate things: a repair approach and a tool. We encode jGenProg based on the material presented in the GenProg publications. As the code of Astor is publicly available in the platform GitHub, researchers and developers can review and evaluate them, and if required to propose improvements if they spot inconsistencies with the original approach.

External validity. We have evaluated the repair approaches built over Astor on 357 buggy program revisions, in five unique software systems, from the Defects4J benchmark (Just et al., 2014). One threat is that the number of bugs may not be large enough nor not representative. To mitigate this threat, we have executed repair approaches from Astor over other datasets of Java bugs (e.g., Ye et al., 2018), and we have found that Astor is also capable to repair bugs from an alternative dataset.

Summary. jGenProg has random components such as the selection of suspicious statements to modify. It is possible that different runs of jGenProg produce different patches. For this reason, we have executed each repair attempt with at least three different random seeds. This is not meant to be a comprehensive solution to randomness. However, our goal is to validate our implementation, not the core idea, which was validated in the original publications. Despite this small number of seeds, our experiments are computationally expensive due to the large number of bugs and the combinatorial explosion of repair approaches and configuration parameters. For instance, the experiment presented in Section 5.4.2 consists of 19,949 trials spanning 2,616 days of computation time (White et al., 2017). Moreover, as studied by different works (Qi et al., 2015a; Smith et al., 2015; Le et al., 2018; Yu et al., 2018; Xin and Reiss, 2017a; Martinez et al., 2016), automated generated patches can suffer from *Overfitting*. Those patches are correct with respect to the test suites used for validating them (they are *test-suite adequate patched*), but yet incorrect. One of the reasons of accepting those patches is that the specification used by a repair approach (such as test-suites in the cases of test-suite based repair approaches) can be incomplete: a test suite does not include any input that triggers the unexpected -and incorrect- behaviour of a patch. In our previous study, we manually analyzed patches generated by approaches built Astor, found that a portion of them overfit the evaluation test suites. However, we believe that the overfitting problem affects to the repair approaches rather than the repair framework itself. Astor provides to developers extensions points for designing a new generation of repair approaches that aim at reducing the overfitting.

8. Conclusion

In this paper we presented Astor, a novel framework developed in Java that encodes the design space of generate-and-validate program repair approaches. The framework contains the implementation of 6 repair approaches. It uniquely provides extension points for facilitating research in the field. The built-in repair approaches provided by Astor have already been used by researchers during the evaluation of their new repair approaches. Moreover, researchers have already implemented new components for Astor's extension points. This paper presented an evaluation of the approaches provided by Astor, which repair 98 real bugs from the Defects4J dataset. We hope that Astor will facilitate the construction of new repair approaches and comparative evaluations in future research in automatic repair. Astor is publicly available at https://github.com/SpoonLabs/astor.

References

Abreu, R., Zoeteweij, P., van Gemund, A.J.C., 2006. An evaluation of similarity coefficients for software fault localization. In: Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing, PRDC '06, pp. 39–46.

Arcuri, A., 2011. Evolutionary repair of faulty software. Appl. Soft Comput. 11 (4), 3494–3514.

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, FSE 2014. ACM, New York, NY, USA, pp. 306–317.

Campos, J., Riboira, A., Perez, A., Abreu, R., 2012. Gzoltar: an eclipse plug-in for testing and debugging. In: 2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, pp. 378–381.

Chen, L., Pei, Y., Furia, C.A., 2017. Contract-based program repair without the contracts. In: Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017. IEEE Press, Piscataway, NJ, USA, pp. 637–647.

Debroy, V., Wong, W.E., 2010. Using mutation to automatically suggest fixes for faulty programs. In: Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation, ICST '10, pp. 65–74.

DeMillo, R.A., Lipton, R.J., Sayward, F.G., 1978. Hints on test data selection: help for the practicing programmer. Computer 11 (4), 34–41.

Duong, H., Nguyen, T., Qi, D., Roychoudhury, A., Chandra, S., 2013. Semfix: program repair via semantic analysis. In: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13. IEEE Press, Piscataway, NJ, USA, pp. 772–781.

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), pp. 349–358.

Durieux, T., Danglot, B., Yu, Z., Martinez, M., Urli, S., Monperrus, M., 2017. The Patches of the Nopol Automatic Repair System on the Bugs of Defects4J version 1.1.0. Université Lille 1 - Sciences et Technologies Research report hal-01480084.

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, AST '16. ACM, New York, NY, USA, pp. 85–91.

Forrest, S., Nguyen, T.V., Weimer, W., Goues, C.L., 2009. A genetic programming approach to automated software repair. In: Proceedings of the 11th Annual conference on Genetic and evolutionary computation. ACM, pp. 947–954.

Fraser, G., Arcuri, A., 2011. Evosuite: automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. ACM, pp. 416–419.

Goues, C.L., 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 34th International Conference on Software Engineering, ICSE '12. IEEE Press, Piscataway, NJ, USA, pp. 3–13.

Goues, C.L., Holtschulte, N., Smith, E.K., Brun, Y., Devanbu, P., Forrest, S., Weimer, W., 2015. The manybugs and introclass benchmarks for automated repair of c programs. IEEE Trans, Softw.e Eng. 41 (12), 1236–1256.

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

Hamlet, R.G., 1977. Testing programs with the aid of a compiler. IEEE Trans. Softw. Eng. 3 (4), 279–290.

Jiang, J., Xiong, Y., 2017. Can defects be fixed with weak test suites? an analysis of 50 defects from defects4j. arXiv: 1705.04149.

Jones, J.A., Harrold, M.J., Stasko, J., 2002. Visualization of test information to assist fault localization. In: Proceedings of the 24th International Conference on Software Engineering, ICSE '02. ACM, New York, NY, USA, pp. 467–477.

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 International Symposium on Software Testing and Analysis (ISSTA), pp. 437–440. San Jose, CA, USA, July 23–25

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

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

Koza, J.R., 1992. Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, MA, USA.

Le, X.-B.D., Chu, D.-H., Lo, D., Goues, C.L., Visser, W., 2017. Jfix: Semantics-based repair of java programs via symbolic pathfinder. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2017. ACM, New York, NY, USA, pp. 376–379.

Le, X.-B.D., Chu, D.-H., Lo, D., Goues, C.L., Visser, W., 2017. S3: Syntax- and semantic-guided repair synthesis via programming by examples. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017. ACM, New York, NY, USA, pp. 593–604.

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

Le, X.-B.D., Thung, F., Lo, D., Goues, C.L., 2018. Overfitting in semantics-based automated program repair. In: Proceedings of the 40th International Conference on Software Engineering, ICSE '18. ACM, New York, NY, USA. 163–163

Liu, X., Zeng, M., Xiong, Y., Zhang, L., Huang, G., 2017. Identifying patch correctness in test-based automatic program repair. arXiv: 1706.09120.

Liu, Y., 2017. Understanding and generating patches for bugs introduced by third-party library upgrades. Master's thesis.

Long, F., Amidon, P., Rinard, M., 2017. Automatic inference of code transforms for patch generation. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017. ACM, New York, NY, USA, pp. 727–739.

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

Long, F., Rinard, M., 2016. Automatic patch generation by learning correct code. SIGPLAN Not. 51 (1), 298–312.

Martinez, M., Durieux, T., Sommerard, R., Xuan, J., Monperrus, M., 2016. Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. Empir. Softw. Eng. 1–29.

Martinez, M., Monperrus, M., 2013. Mining software repair models for reasoning on the search space of automated program fixing. Empir. Softw. Eng. 1–30.

Martinez, M., Monperrus, M., 2016. Astor: a program repair library for java (demo). In: Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016. ACM, New York, NY, USA, pp. 441–444.

Martinez, M., Monperrus, M., 2018. Ultra-Large Repair Search Space with Automatically Mined Templates: the Cardumen Mode of Astor. SSBSE 65–86.

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, ICSE Companion 2014, pp. 492–495.
Mechtaev, S., Yi, J., Roychoudhury, A., 2015. Directfix: looking for simple program repairs. In: Proceedings of the 37th International Conference on Software Engineering-Volume 1. IEEE Press, pp. 448–458.
Mechtaev, S., Yi, J., Roychoudhury, A., 2016. Angelix: scalable multiline program patch synthesis via symbolic analysis. In: Proceedings of the 38th International Conference on Software Engineering, ICSE '16. ACM, New York, NY, USA, pp. 691–701.
Motwani, M., Sankaranarayanan, S., Just, R., Brun, Y., 2017. Do automated program repair techniques repair hard and important bugs? Empir. Softw. Eng. 1–47.
Perkins, J. H., Kim, S., Larsen, S., Amarasinghe, S., Bachrach, J., Carbin, M., Pacheco, C., Sherwood, F., Sidiroglou, S., Sullivan, G., Wong, W.-F., Zibin, Y., Ernst, M. D., Rinard, M., 2009. Automatically patching errors in deployed software. Pages 87–102.
Qi, Y., Mao, X., Lei, Y., Dai, Z., Wang, C., 2013. Does genetic programming work well on automated program repair? In: Computational and Information Sciences (ICCIS), 2013 Fifth International Conference on. IEEE, pp. 1875–1878.
Qi, Y., Mao, X., Lei, Y., Dai, Z., Wang, C., 2014. The strength of random search on automated program repair. In: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014. ACM, New York, NY, USA, pp. 254–265.
Qi, Y., Mao, X., Lei, Y., Wang, C., 2013. Using automated program repair for evaluating the effectiveness of fault localization techniques. In: Proceedings of the 2013 International Symposium on Software Testing and Analysis, ISSTA 2013. ACM, New York, NY, USA, pp. 191–201.
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, ISSTA 2015. ACM, New York, NY, USA, pp. 24–36.
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, ISSTA 2015. ACM, New York, NY, USA, pp. 24–36.
Saha, R.K., Lyu, Y., Yoshida, H., Prasad, M.R., 2017. Elixir: effective object oriented program repair. In: Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017. IEEE Press, Piscataway, NJ, USA, pp. 648–659.
Smith, E.K., Barr, E.T., Goues, C.L., Brun, Y., 2015. Is the cure worse than the disease? Overfitting in automated program repair. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. ACM, pp. 532–543.
Tanikado, A., Yokoyama, H., Yamamoto, M., Sumi, S., Higo, Y., Kusumoto, S., 2017. New strategies for selecting reuse candidates on automated program repair. In: 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC), Vol. 2, pp. 266–267.
Weimer, W., Forrest, S., Goues, C.L., Nguyen, T., 2010. Automatic program repair with evolutionary computation. Commun. ACM 53 (5), 109.
Weimer, W., Fry, Z.P., Forrest, S., 2013. Leveraging program equivalence for adaptive program repair: models and first results. In: Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on, pp. 356–366.
Weimer, W., Nguyen, T., Goues, C.L., Forrest, S., 2009. Automatically finding patches using genetic programming. In: Proceedings of the 31st International Conference on Software Engineering, ICSE '09, pp. 364–374.
Wen, M., Chen, J., Wu, R., Hao, D., Cheung, S.-C., 2017. An empirical analysis of the influence of fault space on search-based automated program repair.
White, M., Tufano, M., Martinez, M., Monperrus, M., Poshyvanyk, D., 2017. Sorting and transforming program repair ingredients via deep learning code similarities.
Xin, Q., Reiss, S.P. , 2017a. Identifying test-suite-overfitted patches through test case generation. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2017. ACM, New York, NY, USA, pp. 226–236.
Xin, Q., Reiss, S.P. , 2017b. Leveraging syntax-related code for automated program repair. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp. 660–670.
Xiong, Y., Wang, J., Yan, R., Zhang, J., Han, S., Huang, G., Zhang, L., 2017. Precise condition synthesis for program repair. In: Proceedings of the 39th International Conference on Software Engineering, ICSE '17. IEEE Press, Piscataway, NJ, USA, pp. 416–426.
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, (ICSE '18). ACM, New York, NY, USA, pp. 789–799. doi:10.1145/3180155.3180182.
Xuan, J., Martinez, M., Demarco, F., Clment, M., Lamelas, S., Durieux, T., Berre, D.L., Monperrus, M., 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. IEEE Trans. Softw. Eng. 43 (1), 34–55. doi:10.1109/TSE.2016.2560811.
Yang, J., Zhikhartsev, A., Liu, Y., Tan, L., 2017. Better test cases for better automated program repair. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017. ACM, New York, NY, USA, pp. 831–841.
Ye, H., Martinez, M., Durieux, T., Monperrus, M., 2018. A Comprehensive Study of Automatic Program Repair on the QuixBugs Benchmark. arXiv: 1805.03454.
Yokoyama, H., Higo, Y., Kusumoto, S., 2017. Evaluating automated program repair using characteristics of defects. In: 2017 8th International Workshop on Empirical Software Engineering in Practice (IWESEP), pp. 47–52.
Yu, Z., Martinez, M., Danglot, B., Durieux, T., Monperrus, M., 2017. Test case generation for program repair: A study of feasibility and effectiveness. Technical Report, arXiv: 1703.00198.
Yu, Z., Martinez, M., Danglot, B., Durieux, T., Monperrus, M., 2018. Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the nopol repair system. Empir. Softw. Eng.
Yuan, Y., Banzhaf, W., 2017. Arja: automated repair of java programs via multi-objective genetic programming.
Zhang, M., Li, X., Zhang, L., Khurshid, S., 2017. Boosting spectrum-based fault localization using pagerank. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2017. ACM, New York, NY, USA, pp. 261–272.