ASTOR: A Program Repair Library for Java

Matias Martinez
University of Lugano, Switzerland

Martin Monperrus
University of Lille & Inria, France

ABSTRACT

During the last years, the software engineering research community has proposed approaches for automatically repairing software bugs. Unfortunately, many software artifacts born from this research are not available for repairing Java programs. To-reimplement those approaches from scratch is costly. To facilitate experimental replications and comparative evaluations, we present Astor, a publicly available program repair library that includes the implementation of three notable repair approaches (jGenProg2, jKali and jMutRepair). We envision that the research community will use Astor for setting up comparative evaluations and explore the design space of automatic repair for Java. Astor offers researchers ways to implement new repair approaches or to modify existing ones. Astor repairs in total 33 real bugs from four large open source projects.

1. INTRODUCTION

Reducing maintenance costs and improving software quality are two extremely important concerns in software engineering. Software bugs are a threat to the perceived software quality and increase maintenance costs. Recently, software repair approaches have been proposed for repairing bugs in an automated manner. A repair approach is both an algorithm (or workflow) and its realization in a prototype. A major problem for open-science is that those prototypes are hard-wired for a given programming language or are not publicly available.

In this paper, we present Astor (Automatic Software Transformations fOr program Repair). Astor automatically repairs Java program and includes three modes corresponding to three notable repair approaches (their original implementations being for C): GenProg [13], Kali [12] and mutation based repair MutRepair from Debroy et al. [3]. They are called jGenProg2, jKali and jMutRepair. Astor also proposes typical routines for repair such as fault localization or program validation in order to facilitate the implementation of new repair systems for Java.

ISSTA 2016

We envision that the research community will use Astor for setting up comparative evaluations. Researchers will explore the design space of automatic repair by modifying already implemented approaches. They may also build novel repair approaches on top of the basic routines provided by Astor.

jGenProg2, jKali and jMutRepair have unique features. jGenProg2 supports locality-aware repair, an optimization presented in our previous work [8], that aims at decreasing the time to repair bugs. JKali exhaustively explores the repair search space based on code removal. JMutRepair supports the easy addition of new mutation operators.

To evaluate jGenProg2, jKali and jMutRepair, we executed them over a large dataset of real bugs [5]. The result of this experiment shows that 33 out of 224 bug (14.7%) are automatically repaired by Astor. It indicates the correctness and scalability of Astor's code base. Since the code of Astor is open, it can be validated by peers. Bug fixes and extensions by the community are welcome.

> To sum up, our contribution is a publicly available program repair library that includes the implementation of three notable repair approaches (jGenProg2, jKali and jMutRepair). We envision that the research community will use Astor for setting up comparative evaluations and for exploring the design space of automatic repair for Java.

The paper continues as follow: Section 2 summarizes the goal of developing Astor. Section 3 describes Astor and Section 4 presents a short evaluation. Section 5 presents the related work. Finally, Section 6 concludes the paper.

2. GOALS

The goals of developing Astor are the following. *a*) to provide an implementation in Java of notable repair approaches originally provided for other programming languages, *b*) to provide a public implementation of notable repair approaches that are not available, *c*) to provide extension points for easily customizing the repair workflow, for example, for adding a new repair operator, *d*) to allow researchers to build novel repair approaches by reusing the routines Astor provides. Since Astor is publicly available on Github, it welcomes bug fixes and extensions, and most importantly from a scientific viewpoint, peer validation of its code.

3. THREE REPAIR MODES

3.1 Astor Core in a Nutshell

Astor provides one with test-suite based repair [9], a tool that takes as input a buggy program, its test suite with at least one failing test case, and produces (when it is possible) a patch that repairs the bug (i.e., all test cases pass after repair). Astor is a meta-repair tool, it proposes several "modes", where each mode corresponds to different repair algorithms. In this section, we present the three main modes of Astor.

3.2 jGenProg2

jGenProg2 is a Java implementation of GenProg [13], a redundancy-based repair approach [8], which synthesizes candidate fixes from existing code written somewhere else in the system under repair. GenProg is for repairing C code, jGenProg2 is for repairing Java code.

3.2.1 Main features of jGenProg2

The design of jGenProg2 is as follows: *a*) Fault localization: it uses an existing fault localization technique called Ochiai [1] and not the original ad hoc metric of GenProg[13]. *b*) Representation: it works at the level of statements: that is, it removes, replaces, and inserts statements. *c*) Navigation of the search space: Astor provides two options, one is evolutionary optimization (stacking several changes), the other is one-point repair consisting of searching for repairs that consist of a single change (no crossover). *d*) Code transformation: the replace operator replaces one statement by another of the same type (e.g. assignment is only replaced by another assignment). *e*) Location awareness: 3 strategies for reusing code: *Application* (GenProg), *Package* and *File* (introduced by Astor, see Section 3.2.2).

3.2.2 Locality aware repair

In this paper we focus on new feature that jGenProg2 introduces: *locality aware repair*.

In the original GenProg version and its subsequent versions, the 'Insert' and 'Replace' operators reuse code that *"is taken exclusively from elsewhere in the same program"* [6]. In other words, the whole application under repair acts as a pool of repair ingredients. Our intuition, based on findings from our previous research [8], is that using smaller ingredient spaces (such as the same file only) would allow jGenProg2 to: 1) find patches faster, 2) find more patches in certain cases. It is here where redundancy-based repair approaches faces a trade-off: including more ingredients into the space would increase the possibility to find a correct repair, but the time to find it into the space would also increase.

We enrich jGenProg2 with the notion of location-aware search space. The *scope* of an ingredient space defines the places where ingredients are taken from. In jGenProg2, there are three scopes: $file$, *package* and *application*. Given an operator *op* that affects statement s from file f, which belongs to package p: *a*) the file scope takes ingredients only from those in file f, *b*) the package scope takes only from those files from package p, *c*) the application scope is the default one, the one of GenProg, it considers the whole codebase

> jGenProg2 is an implementation of GenProg for Java, augmented with location awareness.

3.3 jKali

In [12], Qi et al. have presented Kali, a system that aims at identifying weak test suites and under-specified bugs. Kali performs "repair" by only removing or skipping code of C programs. We have implemented jKali, a Java implementation of Kali built over the Astor primitives. Kali is for C, jKali is for Java code.

jKali carries out an exhaustive search of the repair space, and implements all Kali operators, which are: a) removal of statements, b) change of if conditions to $true$ and $false$, c) insertion of return statements. jKali navigates the suspicious statements retrieved by fault localization: for each of them in descending order of suspiciousness, the tool applies each operator to produce a candidate program. In other words, the exhaustive search is done both on the space of suspicious statements and the space of code transformations (the three ones presented above).

> jKali is a faithful implementation of Kali [12] for Java.

3.4 jMutRepair

Debroy and Wong [3] have devised a repair approach that applies operators taken from mutation testing for repairing C code. We implemented their approach, which we call jMutRepair. jMutRepair applies mutation operators on suspicious *if condition* statements. jMutRepair performs one single change to the condition.

Our implementation considers three kinds of mutation operators: Relational, Logical and Unary, corresponding to operators OP2, OP3 and OP8 of the original paper [3]. For the Relational category, there are six interchangeable operators: $>, >=, <, =<, ==$, and $!=$. For the Logical category, there are two compatible ones: OR, AND. For the Unary category, there are two mutations: *negation* and *positivation* (removal of the negation operator). jMutRepair also carries out an exhaustive search in the solution space and uses test cases for validating the candidate repairs.

> jMutRepair is a mutation-based repair approach implementation for Java with a 3 built-in mutation operators and an easy way to add new ones.

3.5 Implementation

Astor uses Spoon [11] for Java code analysis and manipulation. The code and documentation of Astor are publicly available on GitHub: `https://github.com/SpoonLabs/astor`. Astor has 9.2k loc and 126 Java classes.

4. EVALUATION

We have used Astor for repairing real bugs of open-source Java programs. We present the results of three experiments made on bugs from the Defects4J benchmarks [4].

4.1 Defects4J

We have used the tree modes from Astor for a large evaluation consisting in searching for patches for 224 bugs from the Defects4J benchmarks [5], those from projects Apache Commons Math, Apache Commons Lang, JFreeChart and Joda Time. A patch is said to be test-adequate if it passes all tests, incl. the failing one. As shown by previous work [12], a patch may be test-adequate yet incorrect, when it only works on the inputs from the test suite and does not

Table 1: Test-Adequate Patches for 224 Bugs in Defects4J with the Three Repair Modes of Astor. In Total, 33 bugs (14.7%) are repaired

Project	jGenProg	jKali	jMutRepair
Chart	C1, C3, C5, C7, C13, C15, C25,	C1, C5, C13, C15, C25, C26	C1, C7, C25, C26
	$\sum = 7$	$\sum = 6$	$\sum = 4$
Lang	0	0	L27
	$\sum = 0$	$\sum = 0$	$\sum = 1$
Math	M2, M5, M7, M8, M28, M40, M49, M50, M53, M60, M70, M71, M73, M78, M80, M81, M82, M84, M85, M95	M2, M8, M28, M32, M40, M49, M50, M78, M80, M81, M82, M84, M85, M95	M2, 28, 40, 50, 57, 58, 81, 82, 84, 85, 88
	$\sum = 20$	$\sum = 14$	$\sum = 11$
Time	T4, T11	T4, T11	T11
	$\sum = 2$	$\sum = 2$	$\sum = 1$
Total	29	22	17

generalize. jGenProg2 uses a time limit of 3 hours. jKali and jMutRepair ran without any time limit and perform an exhaustive search.

Table 1 summarizes the results. jGenProg2 finds test-adequate patches for 7 bugs from Chart project, 20 from Commons Math and 2 from Joda Time (resp 6 bugs from Chart, 14 from Math and 2 from Joda Time for jKali). Moreover, jMutRepair is the only approach from Astor to repair one bug from Apache Lang. To sum up jGenProg2 finds test-adequate patches for 29 out of 224 bugs, jKali 22 and jMutRepair 17. All patches are available at `https://goo.gl/yeLXmo`.

> Astor is capable of finding test-adequate patches for 33 real bugs from large-scale Java programs.

4.2 Locality Aware Repair Evaluation

In this section we present a first evaluation of the optimization introduced by jGenProg2 (Section 3.2.2). We run jGenProg2 to repair Commons Math bugs using the baseline configuration (Application scope) and the optimized modes (File and Package scopes). Due to the stochastic nature of jGenProg2 we launch 20 repair attempts per bug and per scope. We set up a timeout of 3 hours for each attempt. Table 2 presents the median time for finding the first patch using the Insert and Replace operators (those that reuse code for synthesizing patches).

We consider that the "Application" scope is the baseline, because it is the one used in GenProg. Table 2 shows that the median time to find the first patch is in average 11.1, 16.7 and 41.1 minutes using the configuration File, Package and Application (baseline), respectively. This validates the fact that locality-aware repair speeds-up repair. The table also shows the percentage of time saved by the optimization. Compared to the base-line, the File scope allows faster repair for 17 out of 21 bugs. Math-71 cannot be repaired using the baseline configuration, yet can be repaired with File and Package scope.

Interestingly, bug Math-60 could not be repair with the optimized ingredient search space. The reason is that the only valid patch ingredient must be taken from another file (resp package) than the file (resp package) of the buggy statement.

> The location-based optimization allows jGenProg2 to reduce repair time from 41.1 to 11.1 minutes (73%), without hampering repair capability.

Table 2: Experimental results on repairing the bugs of Commons Math project with 3 different ingredient scope: File, Package and Application. The locality awareness of jGenProg2 enables to find a patch faster.

Bug ID	Median time for finding the first patch (in minutes)			Time reduction	
	File	Package	Application	File vs App	Pack vs App
Math-2	9	21.5	31	71%	30.6%
Math-5	5.4	5.3	27.8	80.7%	80.9%
Math-7	27.9	29.3	168.6	83.4%	82.6%
Math-28	26.4	33.4	46.2	42.8%	27.6%
Math-40	23	52.6	31.8	27.6%	-39.6%
Math-44	12.1	10.9	47.1	74.3%	76.8%
Math-49	8.7	20.8	19.7	55.6%	-5.6%
Math-50	4.6	5.6	7.4	38.5%	25.1%
Math-53	2	2.2	86.2	97.7%	97.5%
Math-60	-	-	51.1	- %	- %
Math-70	0.2	0.3	31.3	99.3%	99%
Math-71	4.9	7.6	-	- %	- %
Math-73	0.4	0.5	15.5	97.2%	96.7%
Math-74	-	67.4	12.2	- %	-81.9%
Math-78	2.4	4.1	12.6	80.9%	67.7%
Math-80	10.2	3.7	11	7.3%	66.4%
Math-81	6.6	4.5	3.6	-45.7%	-21%
Math-82	10.3	29.9	116.1	91.1%	74.2%
Math-84	36.2	23.1	46.6	22.2%	50.3%
Math-85	18.3	7	46.6	60.7%	85.1%
Math-95	2.3	4.3	9.9	76.7%	56.5%
Total	11.1	16.7	41.1	58.97 %	45.7 %

4.3 Examples of Real Bugs Repaired

In this section, we present three small case studies of repairing real bugs from the project Apache Commons Math using our tools: jGenProg2, jKali, jMutRepair.

4.3.1 Bug Math-70 repaired by jGenProg2

Our implementation of jGenProg2 correctly repairs the bug Math-70 from Defects4J. As listing 1 shows, the patch generated by jGenProg2 replaces a method invocation by an overridden method. The patch is the same that the patch done by the project's developers.

Listing 1: Patch for bug Math-70

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

4.3.2 Bug Math-50 repaired by jKali

jKali is capable of repairing bug Math-50 from project Math , which was reported in Commons Math's issue tracker as major bug #631. The patch, presented in listing 2, removes an `if` condition and its `then` branch and is identical to the patch provided by the developers.

Listing 2: Patch for bug 50

```
- if (x == x1) {
-     f0 = computeObjectiveValue(x0);    }
```

4.3.3 Bug Math-85 repaired by jMutRepair

jMutRepair generates a patch for bug Math-85, which has been reported in the project issue tracker as issue #280. The patch, presented in listing 3 is identical to the developer's patch and changes a relational operator from >= to >.

Listing 3: Patch for bug Math-85

```
- if (fa * fb >= 0.0 ) {
+ if (fa * fb > 0.0 ) {
```

In this section, we have shown that our implementations of state-of-the-art repair approaches are able to repair real Java bugs, by generating patches that are syntactically equivalent to those manually written by developers.

5. RELATED WORK

In the last few years, several test-suite based repair approaches have been proposed, using different techniques for optimizing the solution search. For example, Arcuri [2] applies co-evolutionary computation to automatically generate bug fixes. GenProg by Weimer [13] applies genetic programming to the AST of a buggy program and generates patches by adding, deleting, or replacing AST nodes. The authors provide a GenProg implementation for repairing C code. Our implementation of GenProg targets Java code. Debroy & Wong [3] 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. The tool is not publicly available. Kali by Qi et al. [12] has recently been proposed to examine the fixability power of simple actions, such as statement removal. As GenProg, Kali targets C code. We have built a Java version of Kali over Astor framework, which includes all transformations proposed by Kali. SemFix by Nguyen et al. [10] is a constraint based repair approach for C . This approach provides patches for assignments and conditions by combining symbolic execution and code synthesis. Nopol by DeMarco et al. [14] 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. SPR [7] defines a set of staged repair operators so as to early discard many candidate repairs that cannot pass the supplied test suite. SPR is publicly available but targets C programs.

6. CONCLUSION

In this paper we have presented Astor, a publicly available repair framework for Java that includes an implementation of GenProg, Kali and mutation-based repair. Those threee repair modes can repair a total of 33 real bugs in a benchmark of real bugs from open-source Java programs. We hope that Astor will facilitate future research and comparative evaluations in automatic repair. Astor is publicly available at `https://github.com/SpoonLabs/astor`.

7. REFERENCES

[1] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. An evaluation of similarity coefficients for software fault localization. In *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, PRDC '06, pages 39–46, 2006.

[2] Andrea Arcuri. Evolutionary repair of faulty software. *Appl. Soft Comput.*, 11(4):3494–3514, June 2011.

[3] Vidroha Debroy and W. Eric Wong. 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, pages 65–74, 2010.

[4] Thomas Durieux, Matias Martinez, Martin Monperrus, Romain Sommerard, and Jifeng Xuan. Automatic Repair of Real Bugs: An Experience Report on the Defects4J Dataset. Technical Report 1505.07002, Arxiv, 2015.

[5] René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A Database of existing faults to enable controlled testing studies for Java programs. In *ISSTA 2014, Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440, San Jose, CA, USA, July 23–25, 2014.

[6] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 3–13, Piscataway, NJ, USA, 2012. IEEE Press.

[7] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 166–178, New York, NY, USA, 2015. ACM.

[8] Matias Martinez, Westley Weimer, and Martin Monperrus. 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, pages 492–495, 2014.

[9] Martin Monperrus. A critical review of "automatic patch generation learned from human-written patches": Essay on the problem statement and the evaluation of automatic software repair. In *Proceedings of the International Conference on Software Engineering*, 2014.

[10] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 772–781, Piscataway, NJ, USA, 2013. IEEE Press.

[11] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A library for implementing analyses and transformations of java source code. *Software: Practice and Experience*, page na, 2015.

[12] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 24–36, New York, NY, USA, 2015. ACM.

[13] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 364–374, 2009.

[14] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime ClÃl'ment, Sebastian Lamelas, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, 2016.

APPENDIX

A. APPENDIX

This appendix demonstrates how to download, configure and run Astor with the version of April 2016.

A.1 Download Astor

To download the code, execute:

```
git clone https://github.com/SpoonLabs/astor
```

Astor is a Maven project[1]. That means, its resources are organized as follows: a) one folder 'src' where the source code is located, b) a file called 'pom.xml' which describe the project metadata such as its dependencies, c) a folder 'target' where compiled code (bytecode) is located after a compilation, d) a folder 'examples' which contains real buggy projects for testing Astor.

A.2 Build Astor

To compile Astor project, run command `mvn build`. To generate a bytecode package (jar), run command `mvn package`. This creates jar "astor-X-dependencies.jar" into folder 'target'. For simplicity, you can rename it to 'astor.jar'.

A.3 Setup Buggy Project

Once Astor package is built, let us focus on the project to repair. Astor contains several buggy projects into the folder 'examples' In this demo we use '/examples/math_70' corresponding to bug MATH-70 from Defects4J. First, compile the code of the buggy application to be repair. If it is a maven project (as MATH-70 is) run `mvn compile`.

As Astor needs to know the directory layout of the project to repair, then, identify the folders where source code and byte code are located. For example, folders 'src' and 'bin', or 'src/java/main' and 'target/java/'.

Finally, execute all tests from the buggy project. If the project is a maven project run `mvn test`. There must be at least one failing test case that exposes the bug that Astor will aim at repairing. Write down the names of the failing test cases for passing later to Astor as argument. For MATH-70 the failing test case is:

'org.apache.commons.math.analysis.solvers.BisectionSolverTest"

A.4 Create Astor Command

Once Astor is built and a buggy project is set up, you must create the command to launch Astor over the buggy project.

The main class of Astor is 'fr.inria.main.evolution.AstorMain' Astor requires some arguments, such as:

`-mode` "execution mode (jgenprog, jkali, jmutrepair)"

`-location` "location of the project to repair"

`-dependencies` "folder with the dependencies (jars) of the application to repair"

`-failing` "failing test case names"

`-jvm4testexecution` "JDK location that Astor uses to execute the application to repair"

Additionally, there are many optional arguments that can be passed to Astor. For example, argument `-srcjavafolder` corresponds to the folder name where source code is located inside the project to repair. By default, Astor uses the value 'src/main/java', which corresponds to the standard directory layout defined for Maven.

[1]https://maven.apache.org/

To see the complete list of arguments and their explanations execute:

```
java -cp astor.jar fr.inria.main.evolution.AstorMain -help
```

Default values of all arguments are stored into the file 'configuration.properties'.

Now, let us present the command line to run jGenProg2 over MATH-70:

```
java  -cp astor.jar fr.inria.main.evolution.AstorMain
-location ./examples/math_70
-mode jgenprog  -scope package -failing
org.apache.commons.math.analysis.solvers.BisectionSolverTest
-dependencies ./examples/libs/junit-4.4.jar
-srcjavafolder /src/java/ -srctestfolder /src/test/
-binjavafolder /target/classes
-bintestfolder /target/test-classes
-flthreshold 0.5 -seed 10 -maxtime 100 -stopfirst true
```

Note that this command line executes Astor in mode jGenProg (`-mode jgenprog`), using a package ingredient scope (`-scope package`, see section 3.2.2). Moreover, the command overrides the directory layout of the buggy project (`-srcjavafolder... -bintestfolder...`). Finally, the argument `-maxtime 100` indicates a timeout of 100 minutes to find a patch and `-stopfirst true` means to stop finding more solutions after finding the first one.

A.5 Astor Execution and Results

To execute Astor, it is necessary to install Java JDK 1.8. After the execution of a command, Astor writes in the output folder (property 'workingDirectory' in the configuration file, by default is './outputMutation/') a folder with all the variants generated by an approach. Each variant folder contains the files that Astor has produced and possibly a candidate patch.

The summary of the execution is printed on the screen at the end of the execution. If there is at least one solution, it prints "`Solution found`" and then it lists the program variants that are solution (i.e., repaired versions of the program). Then, the folder of each of those solution variants contains a file "Patch.xml", which summarizes the changes done on the variant for repairing the bug. Listing 4 presents the file generated after executing the presented command line for MATH-70. It is only present if the variant is a valid solution (fixes the failing test and no regression). If Astor does not find any solution in the execution, it prints to the screen "`No solution found`".

Listing 4: Patch.xml for MATH-70

```
<patch>
 <operation generation="42" line="72"
  location=
  "org.apache.commons.math.analysis.solvers.BisectionSolver"
  type="ReplaceOp">
  <original>return solve(min, max)</original>
  <modified scope="LOCAL">return solve(f, min, max)</modified>
 </operation>
</patch>
```

A.6 Using Extension Points

Astor offers extension points to customize existing approaches or to implement new ones. For instance, Astor offers an extension point to add new repair operators. For example, one can include to jMutRepair a new operator that mutates right-hand side expressions of assignments. A new operator class must extend the abstract class 'AstorOperator'. Then, the canonical name of this operator's class is passed to Astor via the argument `-customop`.