Empirical Review of Java Program Repair Tools

A Large-Scale Experiment on 2,141 Bugs and 23,551 Repair Attempts

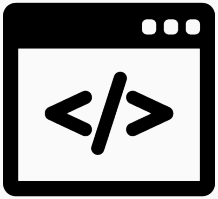
Thomas Durieux, Fernanda Madeiral, Matias Martinez, Rui Abreu

August 28, 2019

INESC-ID, Lisbon, Portugal, Federal University of Uberlândia, Brazil, Université Polytechnique Hauts-de-France

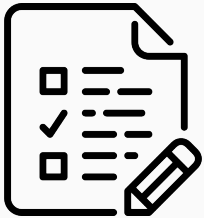
Automatic Patch Generation

Test-based Automatic Patch Generation



Buggy Program

Repair Tool:
GenProg[1],
Nopol[2], Arja[3], ...



Oracle
Test-suite:
- Passing Tests
- Failing Tests

[1] Le Goues et al., "GenProg: A generic method for automatic software repair", *TSE'12*

[2] Xuan et al., "Nopol: Automatic repair of conditional statement bugs in Java programs", *TSE'16*

[3] Yuan and Banzhaf, "ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming", *TSE'18*

Test-based Automatic Patch Generation

Uses the test suite as the specification of the program.

Status	Tests
●	Test Feature 1
●	Test Feature 2
●	Test Feature 3

Test-based Automatic Patch Generation

Uses the test suite as the specification of the program.

Common practice: Developer reproduces a bug with a test

Status	Tests
●	Test Feature 1
●	Test Feature 2
●	Test Feature 3
●	**Reproduced Bug-X**

Test-based Automatic Patch Generation

Uses the test suite as the specification of the program.

Goal: Patch generation techniques make all the tests passing

Status	Tests
●	Test Feature 1
●	Test Feature 2
●	Test Feature 3
●	**Reproduced Bug-X**

How Are Automatic Program Repair Tools Evaluated?

Evaluation Workflow

1. Select a benchmark of bugs
2. Execute the repair tool on each bug
3. Count the number of patched bugs
4. Evaluate the correctness (sometimes)

How Are Automatic Program Repair Tools Evaluated?

Evaluation Workflow

1. Select a benchmark of bugs
2. Execute the repair tool on each bug
3. Count the number of patched bugs
4. Evaluate the correctness (sometimes)

22/24 Java repair tools were evaluated on (a subset of) bugs from **Defects4J**.

Goal

1. Evaluate the repairability of the tools on different benchmarks.
2. Create a repair framework to simplify the repair tools execution and evaluation.

Study Design of RepairThemAll

1. Select benchmarks of bugs
2. Select repair tools
3. Build the repair framework
4. Execute and analyze the results

1. Benchmark Selection

Inclusion criteria: All benchmarks of Java bugs with a test-suite.

Benchmark	# Projects	# Bugs	LOC (Java)
Bears	72	251	62,597
Bugs.jar	8	1158	212,889
Defects4J	6	395	129,592
IntroClassJava	6	297	230
QuixBugs	40	40	190
Total	130	2,141	146,428

2. Java Repair Tool Selection

24 considered automatic repair tools.

1. ACS	9. Jaid	17. SimFix
2. ARJA	10. jGenProg	18. SketchFix
3. CapGen	11. jKali	19. SOFix
4. Cardumen	12. jMutRepair	20. ssFix
5. DeepRepair	13. Kali-A	21. xPAR
6. Elixir	14. LSRepair	22. DynaMoth
7. GenProg-A	15. PAR	23. Nopol
8. HDRepair	16. RSRepair-A	24. NPEFix

2. Repair Tools Selection

4 inclusion criteria.

C1. Publicly available

C2. Possible to run

C3. Runs on bugs from different benchmarks beyond the one used in its original evaluation

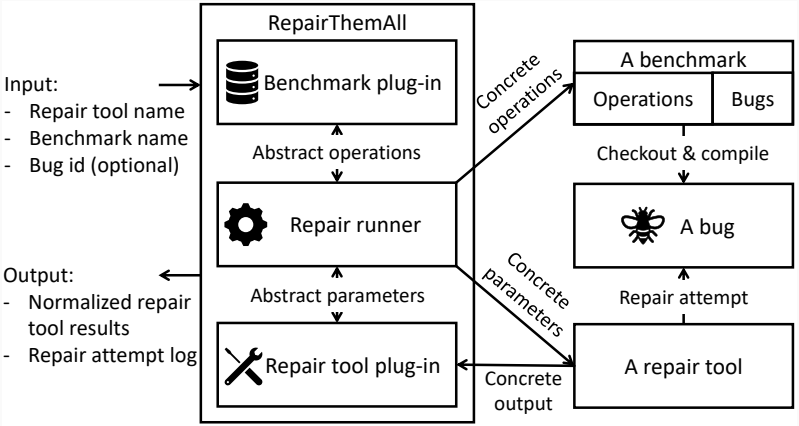
C4. Requires only the source code and the test suite of the program under repair.

2. Repair Tools Selection

11 Selected automatic repair tools.

	Criteria	Repair Tools
Excluded (13)	Not public	Elixir, PAR, SOFix, xPAR
	Not working	ACS, CapGen, DeepRepair
	Multi-bench support	LSRepair, SimFix
	Only source required	HDRepair, Jaid, SketchFix
Included (11)	ARJA, Cardumen, DynaMoth, jGenProg, GenProg-A, jKali, Kali-A, jMutRepair, Nopol, NPEFix, RSRepair-A	

3. Build Repair Framework



Available on GitHub and on Dockerhub.

3. Build Repair Framework

Command line interface

```
python repair.py Nopol --benchmark Defects4J --id Chart-5
```

or

```
docker run tdurieux/repairthemall Nopol --benchmark Defects4J
--id Chart-5
```

Available on GitHub and on Dockerhub.

4. Execution & Analysis

2,141 bugs x **11** tools = **23,551** repair attempts.

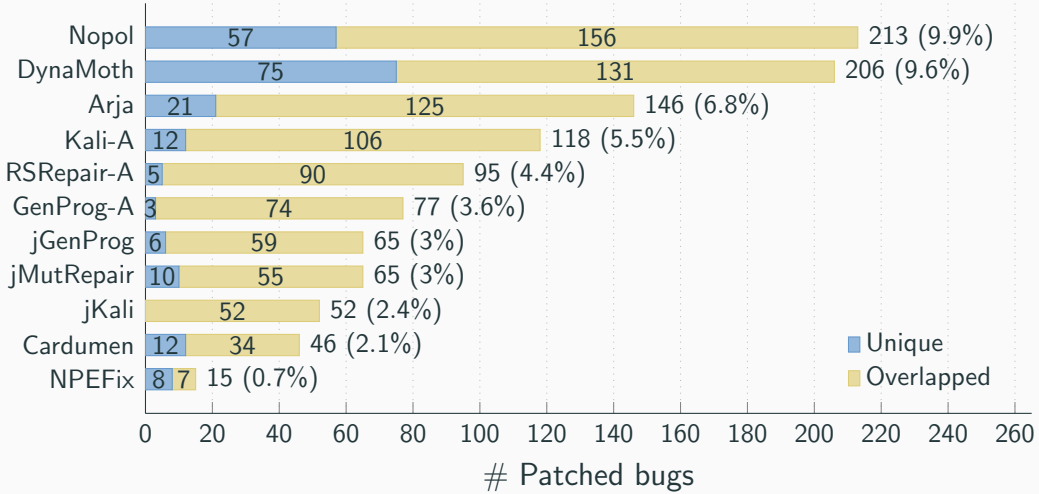
Executed on Grid5000 with a total execution time of 314 days.

Results are available at

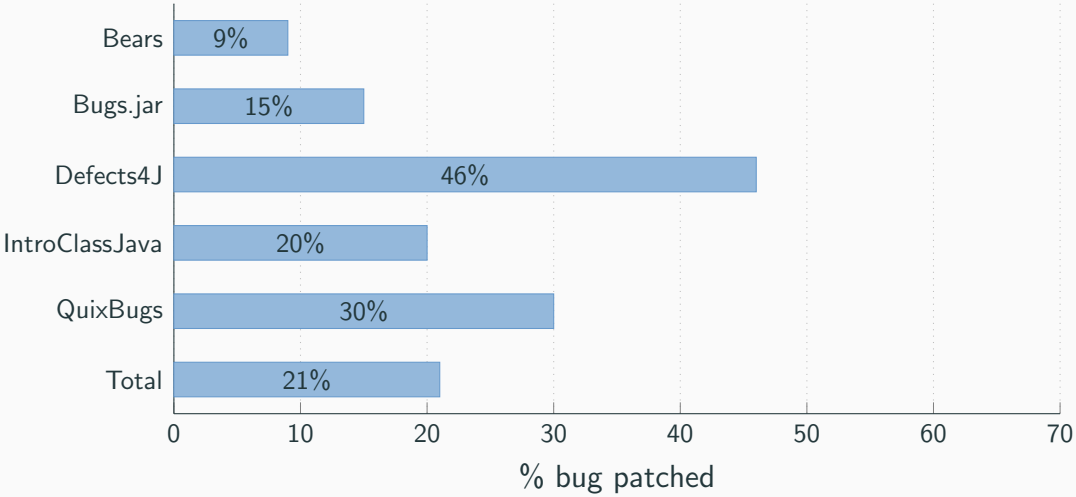
Website:
`program-repair.org/RepairThemAll_experiment/`

Repository:
`github.com/program-repair/RepairThemAll_experiment`

Patched Bugs



Patched Bugs in Benchmarks



Proportion of Patched Bugs

	Defects4J	Other benchmarks	Total	p-value
ARJA	21%	3%	7%	< 0.00001
GenProg-A	11%	2%	3%	< 0.00001
Kali-A	18%	3%	5%	< 0.00001
RSRepair-A	15%	2%	4%	< 0.00001
Cardumen	4%	2%	2%	0.00107
jGenProg	7%	2%	3%	< 0.00001
jKali	6%	1%	2%	< 0.00001
jMutRepair	5%	2%	3%	0.009309
Nopol	27%	10%	11%	< 0.00001
DynaMoth	18%	7%	10%	< 0.00001
NPEFix	2%	<1%	<1%	0.000031

Null hypothesis for benchmark overfitting

The null hypothesis

The number of patched bugs by a given tool is independent of Defects4J.

The null hypothesis

The null hypothesis is violated for all the tools, the number of generated patches is dependent of Defects4J.

Implications & Discussion

Attention at the generalization of results.

What can impact the repairability of tools?

1. Controlled environment: the tools do not handle the complexity of the diversity of the environments.
2. The type of bugs and projects.
3. The bug fix (patch) isolation on `Defects4J`.

Reasons of non-patch generation

1. The repair tool cannot repair the bug
2. Incorrect fault localization
3. Multiple fault locations
4. Small time budget
5. Incorrect configuration
6. Other technical issues

Summary

Take away

Use a diversity of benchmarks to evaluate your automatic repair tools

Contributions

- An Investigation on the **benchmark overfitting**: we have shown that this is a problem
- The RepairThemAll framework
- **66,596** test-suite adequate patches

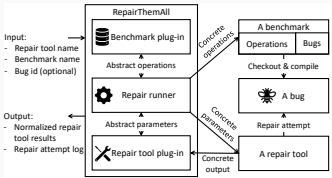
Open-science

All the artifacts are open-science and available on GitHub:

`https://github.com/program-repair/RepairThemAll`

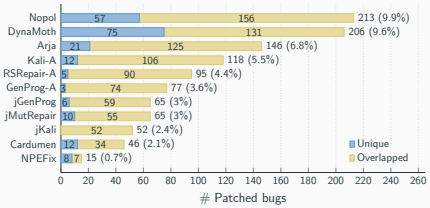
`https://github.com/program-repair/RepairThemAll_experiment`

3. Build Repair Framework



Available on GitHub and on Dockerhub.

Patched Bugs



Proportion of Patched Bugs

	Defects4J	Other benchmarks	Total	p-value
ARJA	21%	3%	7%	< 0.00001
GenProg-A	11%	2%	3%	< 0.00001
Kali-A	18%	3%	5%	< 0.00001
RSRepair-A	15%	2%	4%	< 0.00001
Cardumen	4%	2%	2%	0.00107
jGenProg	7%	2%	3%	< 0.00001
jKali	6%	1%	2%	< 0.00001
jMutRepair	5%	2%	3%	0.009309
Nopol	27%	10%	11%	< 0.00001
DynaMoth	18%	7%	10%	< 0.00001

Reasons of non-patch generation

1. The repair tool cannot repair the bug
2. Incorrect fault localization
3. Multiple fault locations
4. Small time budget
5. Incorrect configuration
6. Other technical issues