An Empirical Study on the Use of Defect Prediction for Test Case Prioritization

International Conference on Software Testing, Verification and Validation
Xi'an, China
April 22-27 2019

DAVID PATERSON,
UNIVERSITY OF SHEFFIELD

JOSE CAMPOS,
UNIVERSITY OF WASHINGTON

RUI ABREU,
UNIVERSITY OF LISBON

GREGORY M. KAPFHAMMER,
ALLEGHENY COLLEGE

GORDON FRASER,
UNIVERSITY OF PASSAU

PHIL MCMINN,
UNIVERSITY OF SHEFFIELD

Defect Prediction

In software development, our goal is to minimize the impact of faults

If we know that a fault exists, we can use *fault localization* to pinpoint the code unit responsible

If we don't know that a fault exists, we can use ***defect prediction*** to estimate which code units are likely to be faulty

Defect Prediction

ClassA 33%	ClassB 10%
ClassC 72%	ClassD 3%

Defect Prediction

Code Smells

- Feature Envy
- God Class
- Inappropriate Intimacy

Code Features

- Cyclomatic Complexity
- Method Length
- Class Length

Version Control Information

- Number of Changes
- Number of Authors
- Number of Fixes

Why Do We Prioritize Test Cases?

Regression testing can account for up to **80%** of the total testing budget, and up to **50%** of the cost of software maintenance

In some situations, it may not be possible to re-run all test cases on a system

By *prioritizing test cases*, we aim to ensure faults are detected in the **smallest amount of time** irrespective of program changes

How Do We Prioritize Test Cases?

	t_1	t_2	t_3	t_4
Version 1	✓	✓	✓	✗
Version 2	✓	✓	✓	✗
Version 3	✓	✓	✓	✗
Version 4	✓	✓	✓	✗
Version 5	✓	✓	✓	✓
Version 6	✓	✓	✓	✓
Version 7	✓	✓	✗	✓
Version 8	✓	✓	✓	✓
Version 9	✗	✓	✓	✓
Version n	?	?	?	?
Version n+1	?	?	?	?

...

t_{n-3}	t_{n-2}	t_{n-1}	t_n
✓	✓	✓	✓
✓	✓	✓	✓
✓	✓	✓	✓
✗	✓	✓	✓
✓	✓	✓	✓
✓	✗	✓	✓
✓	✗	✓	✓
✓	✗	✓	✓
✓	✓	✓	✓
?	?	?	?
?	?	?	?

DPATERSON1@SHEFFIELD.AC.UK

How Do We Prioritize Test Cases?

This Paper

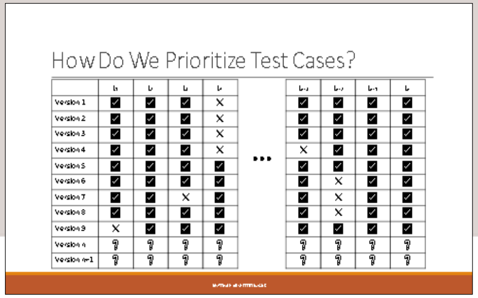
Code Coverage

"How many lines of code are executed by this test case?"

```
public int abs(int x){
    if (x >= 0) {
        return x;
    } else {
        return -x;
    }
}
```

Test History

"Has this test case failed recently?"



Defect Prediction:

"What is the likelihood that this code is faulty?"

Defect Prediction for Test Case Prioritization

ClassA	ClassB	ClassC	ClassD
33%	10%	72%	3%

DPATERSON1@SHEFFIELD.AC.UK

Defect Prediction for Test Case Prioritization

ClassC

72%

Defect Prediction for Test Case Prioritization

ClassC

72%

Test Cases that execute code in ClassC:

- TestClass.testOne
- TestClass.testSeventy
- OtherTestClass.testFive
- OtherTestClass.testThirteen
- TestClassThree.test165

How do we order these test cases before placing them in the prioritized suite?

Secondary Objectives

```
Test Cases that execute code in ClassC:

- TestClass.testOne
- TestClass.testSeventy
- OtherTestClass.testFive
- OtherTestClass.testThirteen
- TestClassThree.test165
```

We can use one of the features described earlier (e.g. code coverage) as a way of ordering the *subset* of test cases

Secondary Objectives

```
Test Cases that execute code in ClassC:   Lines Covered:

- TestClass.testOne                                   25
- TestClass.testSeventy                               32
- OtherTestClass.testFive                            144
- OtherTestClass.testThirteen                          8
- TestClassThree.test165                              39
```

We can use one of the features described earlier (e.g. code coverage) as a way of ordering the *subset* of test cases

Secondary Objectives

```
Test Cases that execute code in ClassC:   Lines Covered:

- OtherTestClass.testFive                            144
- TestClassThree.test165                              39
- TestClass.testSeventy                               32
- TestClass.testOne                                   25
- OtherTestClass.testThirteen                          8
```

We can use one of the features described earlier (e.g. code coverage) as a way of ordering the *subset* of test cases

Defect Prediction for Test Case Prioritization

ClassC

72%

Test Cases that execute code in ClassC:

- OtherTestClass.testFive
- TestClassThree.test165
- TestClass.testSeventy
- TestClass.testOne
- OtherTestClass.testThirteen

Prioritized Test Suite:

Defect Prediction for Test Case Prioritization

ClassC

72%

Test Cases that execute code in ClassC:

Prioritized Test Suite:

- OtherTestClass.testFive
- TestClassThree.test165
- TestClass.testSeventy
- TestClass.testOne
- OtherTestClass.testThirteen

Defect Prediction for Test Case Prioritization

ClassA

33%

Test Cases that execute code in ClassA:	Lines Covered:
- ClassATest.testA	14
- ClassATest.testB	27
- ClassATest.testC	9

Prioritized Test Suite:

- OtherTestClass.testFive
- TestClassThree.test165
- TestClass.testSeventy
- TestClass.testOne
- OtherTestClass.testThirteen

Defect Prediction for Test Case Prioritization

ClassA

33%

Test Cases that execute code in ClassA:	Lines Covered:
- ClassATest.testB	27
- ClassATest.testA	14
- ClassATest.testC	9

Prioritized Test Suite:

- OtherTestClass.testFive
- TestClassThree.test165
- TestClass.testSeventy
- TestClass.testOne
- OtherTestClass.testThirteen

Defect Prediction for Test Case Prioritization

ClassA

33%

Test Cases that execute code in ClassA:

Prioritized Test Suite:

- OtherTestClass.testFive
- TestClassThree.test165
- TestClass.testSeventy
- TestClass.testOne
- OtherTestClass.testThirteen
- ClassATest.testB
- ClassATest.testA
- ClassATest.testC

Defect Prediction for Test Case Prioritization

By repeating this process for all classes in the system, we generate a fully prioritized test suite based on defect prediction

Empirical Evaluation

Empirical Evaluation

Defect Prediction: Schwa [1]

Uses version control information to produce defect prediction scores comprised of weighted number of commits, authors, and fixes related to a file

[1] - https://github.com/andrefreitas/schwa

Empirical Evaluation

Defect Prediction: Schwa [1]

Uses version control information to produce defect prediction scores comprised of weighted number of commits, authors, and fixes related to a file

Faults: DEFECTS4J [2]

Repository containing 395 real faults collected across 6 open-source Java projects

[1] - https://github.com/andrefreitas/schwa
[2] - https://github.com/rjust/defects4j

Empirical Evaluation

Defect Prediction: Schwa [1]

Uses version control information to produce defect prediction scores comprised of weighted number of commits, authors, and fixes related to a file

Faults: DEFECTS4J [2]

Repository containing 395 real faults collected across 6 open-source Java projects

Test Prioritization: KANONIZO [3]

Test Case Prioritization tool built for Java Applications

[1] - https://github.com/andrefreitas/schwa
[2] - https://github.com/rjust/defects4j
[3] - https://github.com/kanonizo/kanonizo

1

Discover the best parameters for defect prediction in order to predict faulty classes as soon as possible

2

Compare our approach against existing **coverage-based** approaches

3

Compare our approach against existing **history-based** approaches

Research Objectives

Parameter Tuning

1. Revisions Weight
2. Authors Weight
3. Fixes Weight
4. Time Weight

$$\sum \boldsymbol{RevisionsWeight} + \boldsymbol{AuthorsWeight} + \boldsymbol{FixesWeight} = \mathbf{1}$$

Parameter Tuning

1

$$\sum RevisionsWeight + AuthorsWeight + FixesWeight = 1$$

Revisions Weight	Authors Weight	Fixes Weight	Time Range
1.0	0.0	0.0	0.0
0.9	0.1	0.0	0.0
0.8	0.2	0.0	0.0
		.	
		.	
		.	
0.0	0.0	1.0	0.9
0.0	0.0	1.0	1.0

726 Valid Configurations

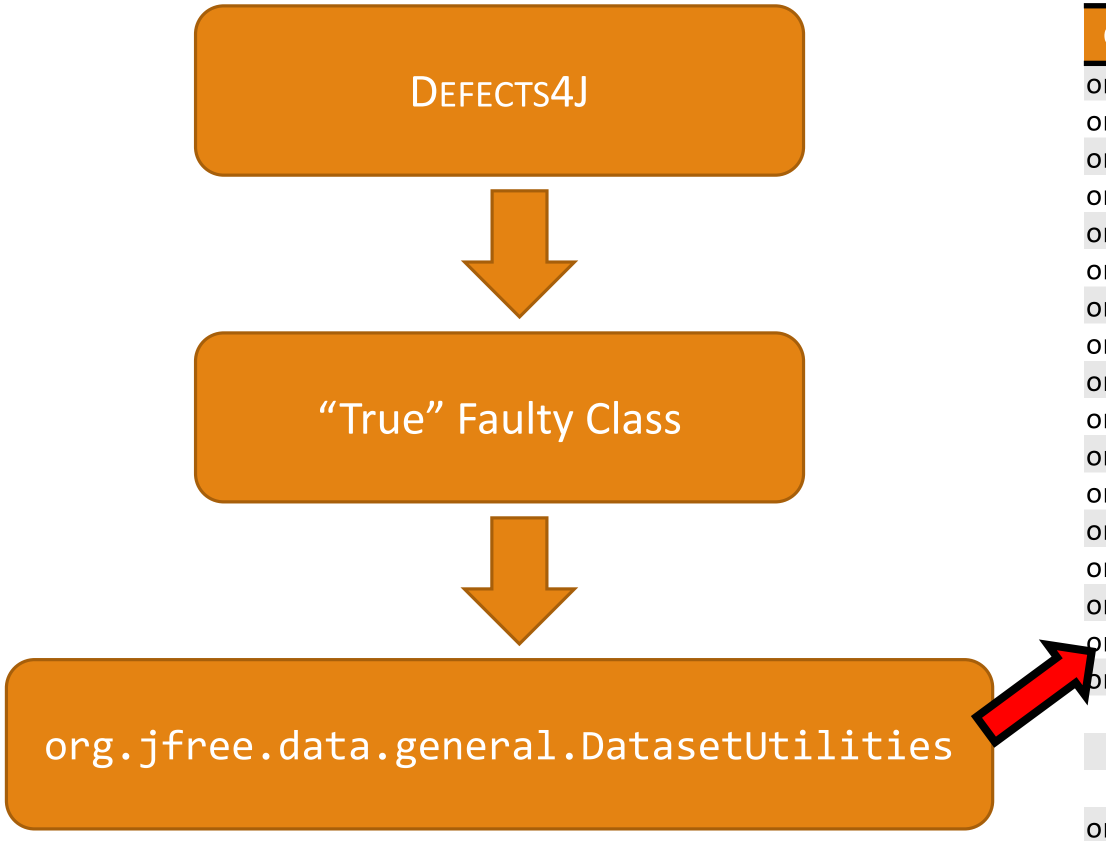
Parameter Tuning

- Select 5 bugs from each project at random
- For each bug/valid configuration
  - Initialize Schwa with configuration and run
  - Collect “true” faulty class from DEFECTS4J
  - Calculate index of “true” faulty class according to prediction

Parameter Tuning

Class Name	Prediction
org.jfree.chart.plot.XYPlot	99.98
org.jfree.chart.ChartPanel	99.92
org.jfree.chart.renderer.xy.AbstractXYItemRenderer	99.30
org.jfree.chart.plot.CategoryPlot	99.20
org.jfree.chart.renderer.AbstractRenderer	98.58
org.jfree.chart.renderer.category.AbstractCategoryItemRenderer	98.02
org.jfree.chart.renderer.category.BarRenderer	95.82
org.jfree.chart.renderer.xy.XYBarRenderer	95.22
org.jfree.chart.plot.Plot	94.75
org.jfree.data.time.TimeSeriesCollection	94.53
org.jfree.data.xy.XYSeriesCollection	94.48
org.jfree.chart.plot.junit.XYPlotTests	94.35
org.jfree.chart.renderer.category.StatisticalLineAndShapeRenderer	93.80
org.jfree.chart.renderer.xy.XYItemRenderer	92.43
org.jfree.chart.panel.RegionSelectionHandler	92.24
org.jfree.data.general.DatasetUtilities	92.11
org.jfree.chart.axis.CategoryAxis	90.82
+1091 more…	
org.jfree.data.time.junit.TimePeriodValuesTests.MySeriesChangeListener	0.30

Parameter Tuning



Class Name	Prediction
org.jfree.chart.plot.XYPlot	99.98
org.jfree.chart.ChartPanel	99.92
org.jfree.chart.renderer.xy.AbstractXYItemRenderer	99.30
org.jfree.chart.plot.CategoryPlot	99.20
org.jfree.chart.renderer.AbstractRenderer	98.58
org.jfree.chart.renderer.category.AbstractCategoryItemRenderer	98.02
org.jfree.chart.renderer.category.BarRenderer	95.82
org.jfree.chart.renderer.xy.XYBarRenderer	95.22
org.jfree.chart.plot.Plot	94.75
org.jfree.data.time.TimeSeriesCollection	94.53
org.jfree.data.xy.XYSeriesCollection	94.48
org.jfree.chart.plot.junit.XYPlotTests	94.35
org.jfree.chart.renderer.category.StatisticalLineAndShapeRenderer	93.80
org.jfree.chart.renderer.xy.XYItemRenderer	92.43
org.jfree.chart.panel.RegionSelectionHandler	92.24
org.jfree.data.general.DatasetUtilities	92.11
org.jfree.chart.axis.CategoryAxis	90.82
+1091 more…	
org.jfree.data.time.junit.TimePeriodValuesTests.MySeriesChangeListener	0.30

Position: 16

Parameter Tuning

Revisions are important – best results were observed when revisions weight was **high**

TOP 3:

Revisions Weight	Authors Weight	Fixes Weight	Time Range	Average Position
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
0.7	0.1	0.2	0.4	49.49
[illegible]	0.1	0.3	0.4	49.26

No single configuration significantly outperformed all others

Author Weight should be low – this indicates that the number of authors has little impact

Fixes weight is similar in both

BOTTOM 3:

Revisions Weight	Authors Weight	Fixes Weight	Time Range	Average Position
[illegible]	0.6	0.3	1.0	88.07
0.1	0.7	0.2	1.0	[illegible]
0.1	0.8	0.1	1.0	[illegible]

The 3 **worst** results all occurred when the time range was 1 – this indicates that newer commits are more important to analyze

1

Parameter Tuning

Project	Top 1	Top 1%	Top 5%	To
Chart	1	7	14	[illegible]
Closure	1	31	77	107
Lang	9	11	26	39
Math	1	15	40	55
ito	3	14	29	33
[illegible]	2	9	14	17
Total	**17**	**87**	**200**	**267**

For 67.5% of the bugs, the faulty class was inside the top 10% of classes

For 17 faults, Schwa predicted the **correct faulty class**

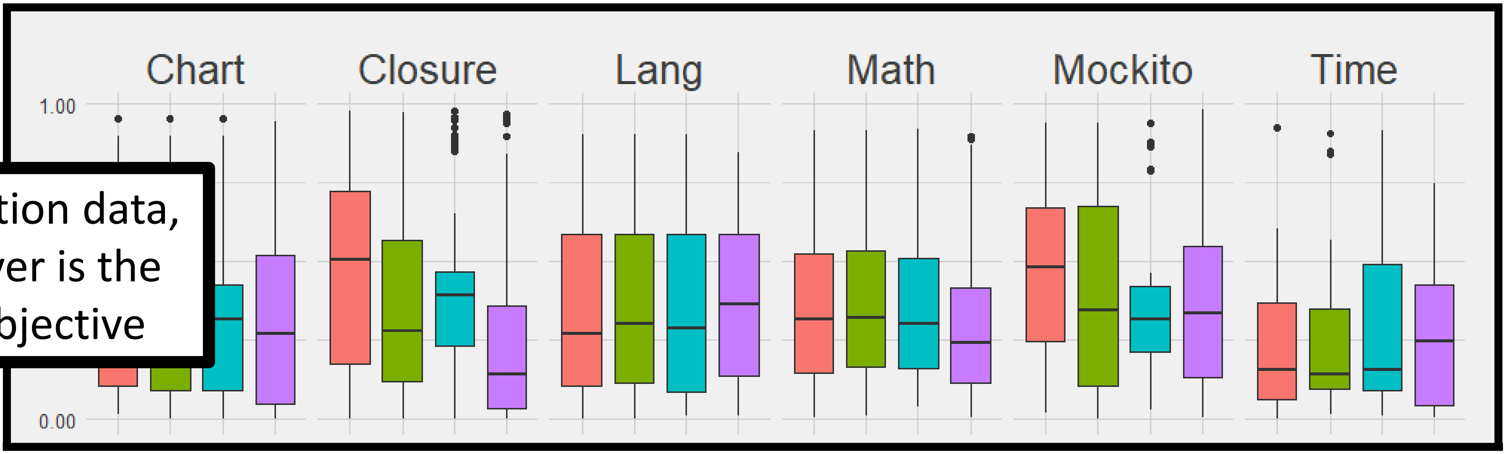
Schwa can effectively predict the location of real faults in DEFECTS4J

Parameter Tuning

1. Greedy
2. Additional Greedy
3. Random
4. Constraint Solver

Parameter Tuning

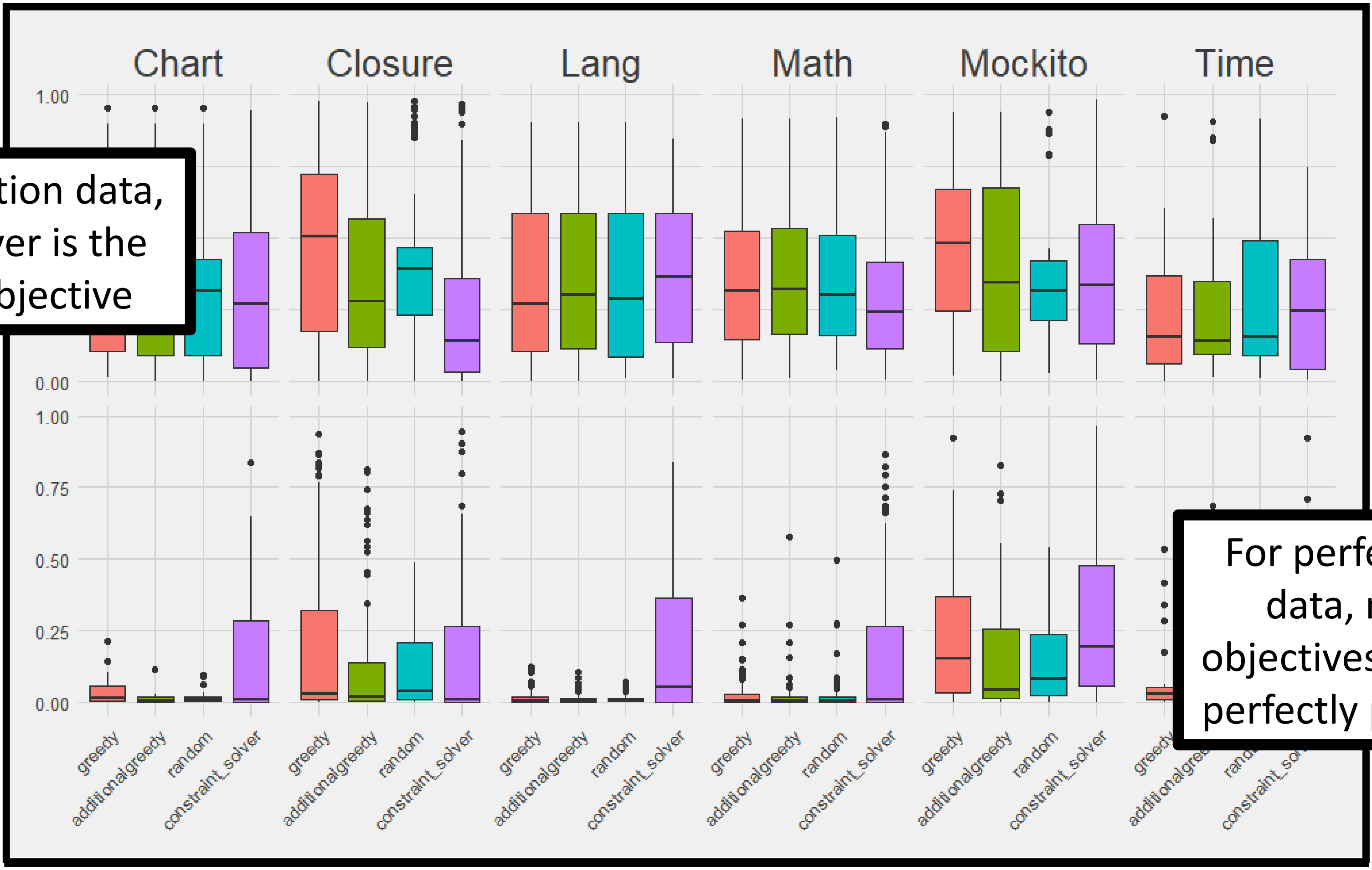
1



For real bug prediction data, the constraint solver is the best secondary objective

Parameter Tuning

1



For real bug prediction data, the constraint solver is the best secondary objective

For perfect bug prediction data, most secondary objectives are able to almost perfectly prioritize test cases

1

Discover the best parameters for defect prediction in order to predict faulty classes as soon as possible

2

Compare our approach against existing **coverage-based** approaches

3

Compare our approach against existing **history-based** approaches

Research Objectives

2

Our Approach vs Coverage-Based

365 faults from DEFECTS4J

5 coverage-based strategies

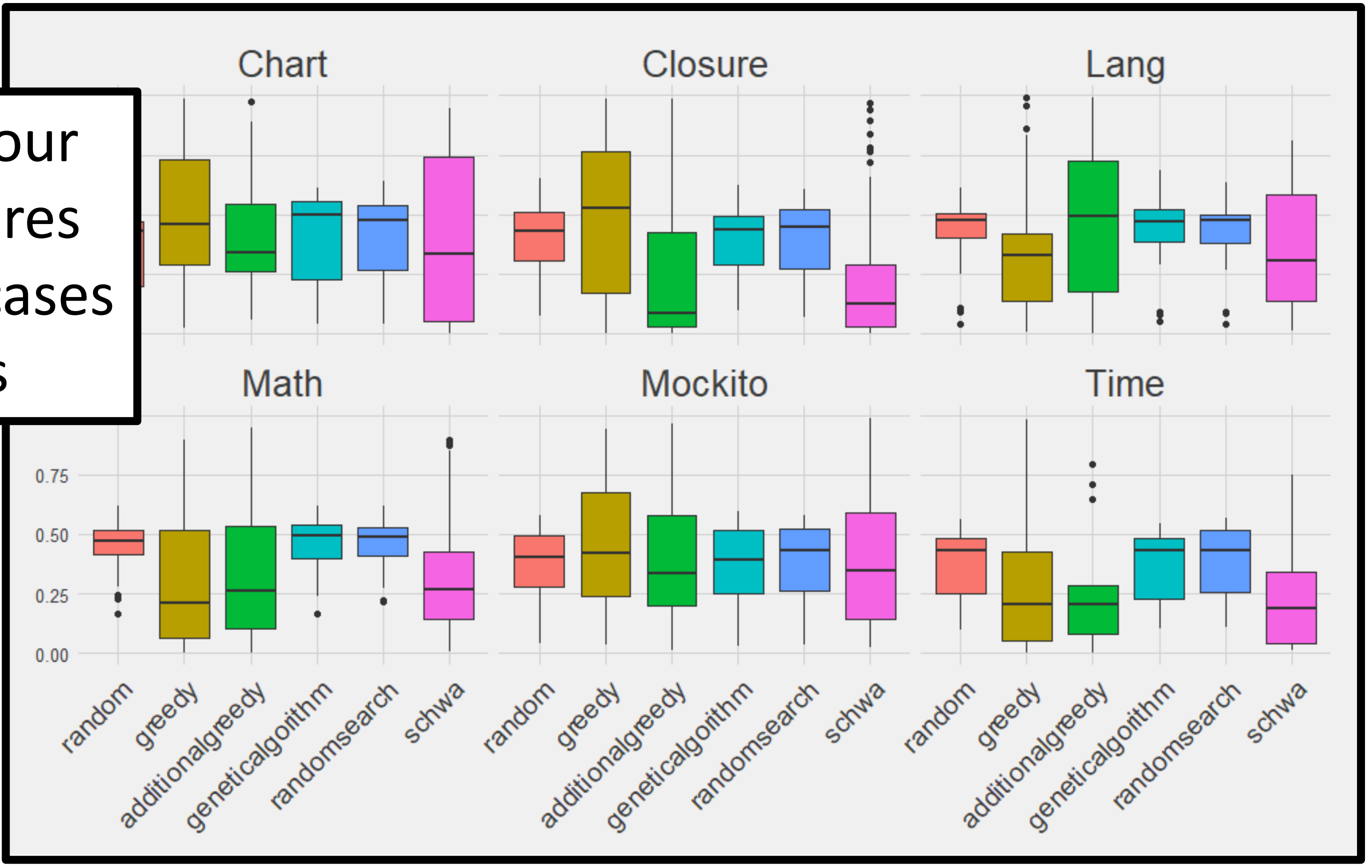
Total 1,825 combinations of fault/strategy

Our approach is best for 1,165 combinations

Significantly outperforms 4 of the 5 strategies

Our Approach vs Coverage-Based

In most cases, our approach requires the fewest test cases to find faults



1

Discover the best parameters for defect prediction in order to predict faulty classes as soon as possible

2

Compare our approach against existing **coverage-based** approaches

3

Compare our approach against existing **history-based** approaches

Research Objectives

3

Our Approach vs History-Based

82 faults from DEFECTS4J

4 history-based strategies

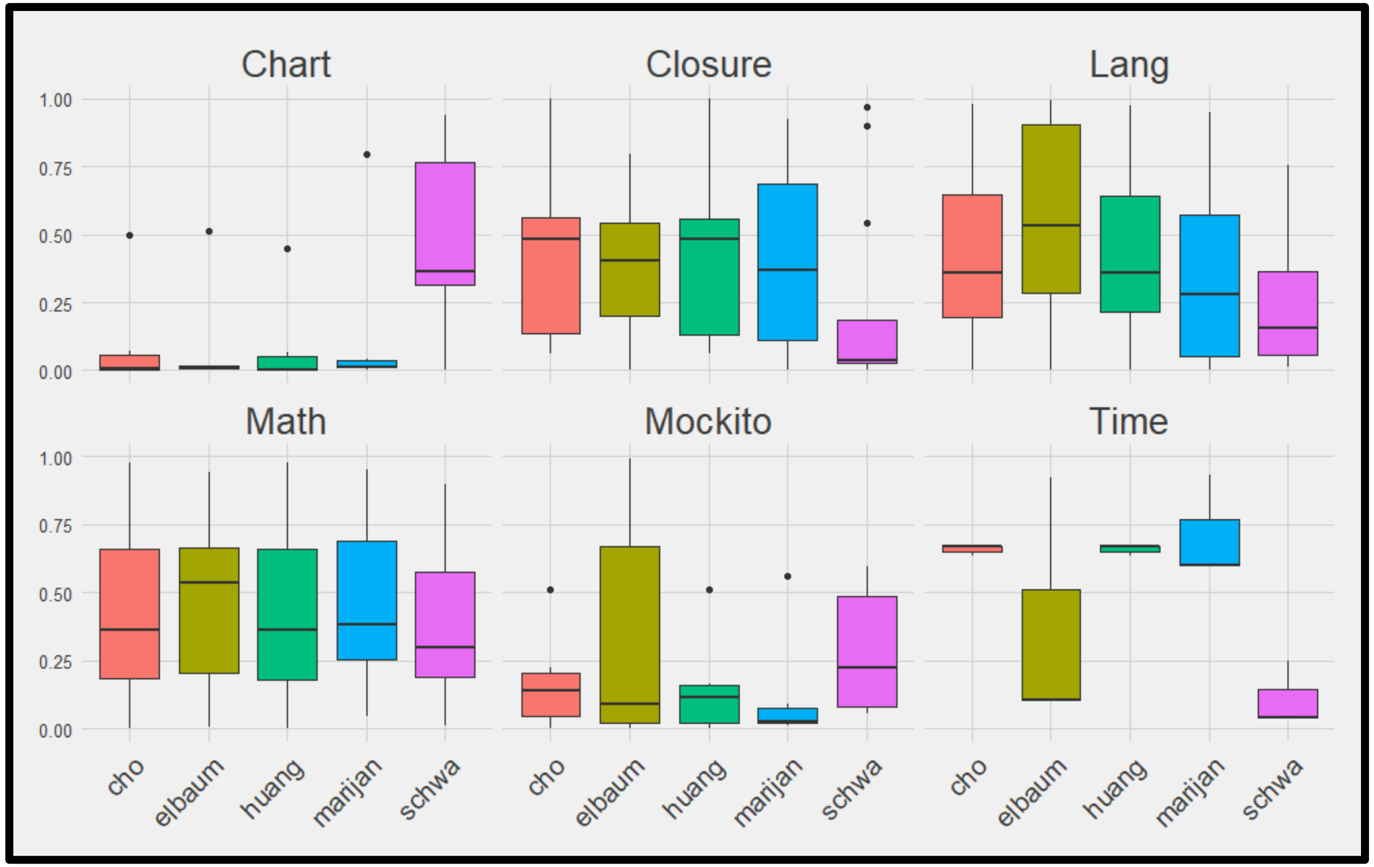
Total 328 combinations of fault/strategy

Our approach is best for 209 combinations

Significantly outperforms 3 of the 4 strategies

3

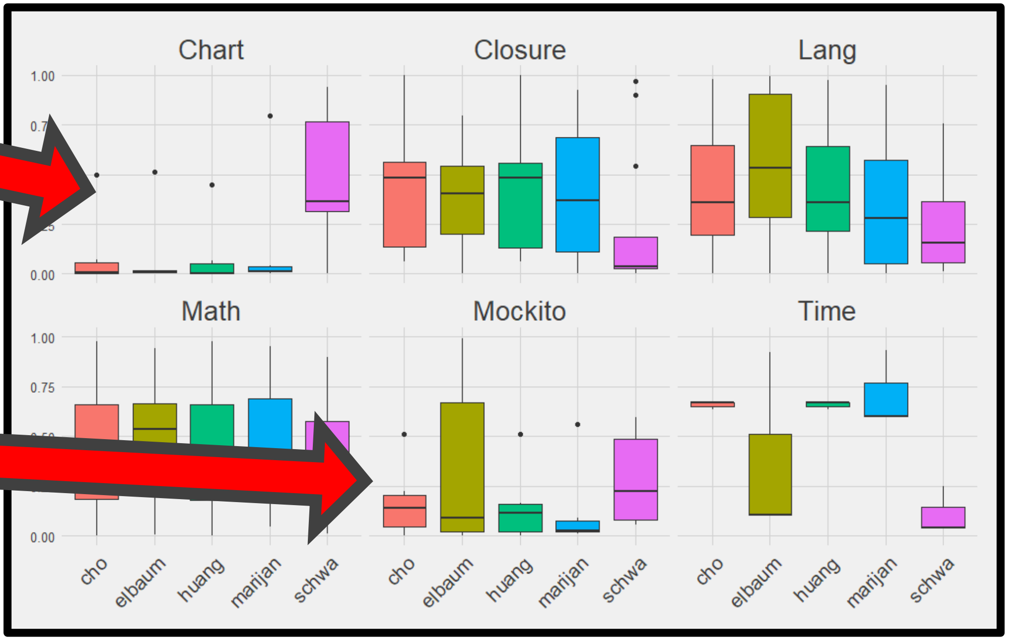
Our Approach vs History-Based



3

Our Approach vs History-Based

Project	Avg. Commits	% Occurrences	Num Failures
Chart	24	73%	**67%**
Closure	178	82%	0%
Lang	159	87%	5%
Math	383	77%	6%
Mockito	105	65%	**19%**
Time	36	100%	0%



Summary

Defect Prediction

- In software development, our goal is to minimize the impact of faults
- If we know that a fault exists, we can use *fault localization* to pinpoint the code unit responsible
- If we don't know that a fault exists, we can use ***defect prediction*** to estimate which code units are likely to be faulty

Why Do We Prioritize Test Cases?

- Regression testing can account for up to **80%** of the total testing budget, and up to **50%** of the cost of software maintenance
- In some situations, it may not be possible to re-run all test cases on a system
- By *prioritizing test cases*, we aim to ensure faults are detected in the **smallest amount of time** irrespective of program changes

Research Objectives

1. Discover the best parameters for defect prediction in order to predict faulty classes as soon as possible
2. Compare our approach against existing **coverage-based** approaches
3. Compare our approach against existing **history-based** approaches

DPATERSON1@SHEFFIELD.AC.UK

3 Our Approach vs History-Based

- 82 faults from DEFECTS4J
- 4 history-based strategies
- Total 328 combinations of fault/strategy
- **Our approach is best for 209 combinations**
- **Significantly outperforms 3 of the 4 strategies**

Tool: https://github.com/kanonizo/kanonizo

Data: https://bitbucket.org/josecampos/history-based-test-prioritization-data

DPATERSON1@SHEFFIELD.AC.UK

Constraint Solver

	L_1	L_2	L_3
TC_1	1	0	1
TC_2	0	1	0
TC_3	1	1	0

In order to cover L_1, we must select either TC_1 or TC_3

$$(TC_1 \lor TC_3) \land (TC_2 \lor TC_3) \land (TC_1)$$

Minimal set:

$$(TC_1 \land TC_2)$$
$$(TC_1 \land TC_3)$$

Statistical Tests

For each of our experiments, we calculated:

- The Mann-Whitney U Test *p-value* in order to calculate the likelihood that our results were observed as a result of chance

- The Vargha-Delaney *effect size*, to measure the magnitude of difference between results

- The *ranking position* of each configuration

DPATERSON1@SHEFFIELD.AC.UK