Réparation Automatique des Logiciels: le Rêve et la Fantaisie
GDR GPL, 14 Juin 2021
Prof. Dr. Tegawendé F. BISSYANDE
Background: Education, Mobility, Positions

1. Burkina Faso
   - Born in 1985
   - High school diploma in 2004

2. Morocco
   - 2004-2006 « Classes Préparatoires »

3. France
   - Telecom Engineering degree / 2006-2009
   - PhD in Computer Science (Debugging Linux) / 2009-2013
     @University of Bordeaux

4. United States
   - Engineering Internship (Demosaicing) 2008
     @UC Santa Barbara

5. Singapore
   - Visiting PhD Data mining – 2012
     @Singapore Mgmt. Univ.

6. Luxembourg
   - Research Associate / 2013-2016
   - Research Scientist / since 2016
     @SnT, University of Luxembourg

- Eiffel Excellence Scholarship for « Grandes Ecoles »
- CNRS scholarship for PhD candidates from Developing Countries (BDI – PED)
Agenda

- Intro: me & SnT/Uni.lu
- Program Repair
- Preliminary insights
- Next steps for industry
The University of Luxembourg

The University of Luxembourg is a research university with a distinctly international, multilingual and interdisciplinary character.

The University’s ambition is to provide the highest quality research and teaching in its chosen fields and to generate a positive scientific, educational, social, cultural and societal impact in Luxembourg and the Greater Region.
The University of Luxembourg

Research Focus Areas

- Computer Science & ICT Security
- European and International Law
- Finance and Financial Innovation
- Education
- Materials Science
- Contemporary and Digital History
- Interdisciplinary theme: Health and Systems Biomedicine
- Interdisciplinary theme: Data Modelling and Simulation

3 Faculties

3 Interdisciplinary Centres
Our vision

A leading international research and innovation centre in secure, reliable and trustworthy ICT systems and services. We play an instrumental role in Luxembourg by boosting R&D investments leading to economic growth and highly qualified talent.

Collaborative, demand-driven research model based on strategic partnerships

High-risk long-term research

Interdisciplinary research approach in key economic sectors

Highly selective global recruitment
Key Figures

**PEOPLE**
- 385 workforce
- 66 nationalities
- 36% alumni who stay in Luxembourg

**PARTNERSHIPS & INNOVATION**
- 47% of Doctoral candidates on Industrial projects
- >50 partners
- 5M Partners annual contribution in Euros
- 5 Spin-offs
Breathing Trust into Business-Critical Software

Most (if not all) modern business-critical solutions rely on software.

e.g., E-payment, blockchain-based solutions, machine-learning based approaches, mobile apps, etc.

Critical Questions:

How to foster the development of Trustworthy software-based solutions?

With:

- Quality of service (crash, bug)
- Limited Security Risks (vulnerabilities)
- Accounting for Compliance (GDPR)
Trustworthy Software Engineering

Software Security
- Vulnerability detection, Data Leaks
- GDPR compliance
- Malware Detection, Piggybacking Detection

Software Repair
- Patch Recommendation
- Automated Program Repair
- Bug Detection
- Vulnerability patching

Explainable Software
- Information Retrieval
- Natural Language Processing
- Time Series Pattern Recognition
- Machine learning
Program Repair Task Force: Those who *did* the work!

Hiring PhDs and Postdocs Now →

Kui Lui (now @NUAA)
Anil Koyuncu (now @Sabanci)
Haoye Tian (2 years left)
Kisub Kim (2 weeks left)
Program Repair
Fixing Bugs is Expensive

The cost of correcting bugs during the testing (QA) stage is $7,136.

The cost of correcting bugs after release is $14,102.

Let's Recall Traditional Bug Fixing

Detection

Localization

Generation
From Manual to Automated Fixing

- Test Automation
- Automated Bug/Fault Localization
- Static/Dynamic Analysis
- Program Repair
Automated Program Repair (APR)

- Heuristic-based program repair, e.g., GenProg, SimFix, CapGen, AVATAR.

- Constraint-based program repair, e.g., Nopol, ACS, and Cardumen.

- Learning-aided program repair, e.g., Deepfix, Prophet, and Genesis.

Template-based APR

Fix pattern data base

Selected fix pattern

Modify buggy code with donor code

Patch Candidates

AST of buggy code
Typical generate-and-validate pipeline / “Template-based”

- In practice, when can we identify the fault location?
- In practice, where should we get the patterns?
- In practice, is there a test suite available to validate the generated patch?
Research axes

- How do we localize faults in practice?
- Can we ignore the assumptions of exhaustive test suites?
- Can we be efficient in the generation of patches?
- Can we predict patch correctness beyond tests?
More on this talk


Preliminary Insights
**How could such a comparison be unfair?**

*The numbers in parenthesis(#) denote the number of bugs fixed by APR tools but ignoring the patch ranking.*
What are the assumptions of fault localization?
Limited discussion on the impact of fault localization on APR tool performance.
Variabilities in FL integration within the Tools

<table>
<thead>
<tr>
<th>Fault Localization (FL) techniques integrated into state-of-the-art APR tools.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FL testing framework</strong></td>
</tr>
<tr>
<td>FL ranking metric</td>
</tr>
<tr>
<td>Supplementary information</td>
</tr>
<tr>
<td>Faulty method is known</td>
</tr>
</tbody>
</table>

* The unspecified/unconfirmed information of an APR tools is marked with “?”. If an APR tool does not use any supplementary information for FL, the corresponding table cell is marked with ‘Ø’.

1. APR tools may add some adaptations to the classical FL
2. Unknown to what extent performance is »just » due to better FL
3. Missing FL details for replication/reproduction
Repair Tool Performance Assessment

1. If the testing frameworks are different
2. If the localization assumptions are different
Localizability of benchmark bugs

Localizability:

File Level

Method Level

Line Level

```
--- a/src/org/jfree/data/time/Week.java
+++ b/src/org/jfree/data/time/Week.java
@@ -173,1 +173,1 @@

public Week(Date time, TimeZone zone) {
    // defer argument checking...

-    this(time, RegularTimePeriod.DEFAULT_TIME_ZONE,
       Locale.getDefault());
+
    this(time, zone, Locale.getDefault());
}
```
One third of bugs in the Defects4J dataset cannot be localized at line level by the commonly used automated fault localization tool.
Localizability of Defects4J bugs

Only a fraction of bugs can be localized with high positions in the ranking list of suspicious positions.
Impact of Effective Localization Ranking

APR tools are prone to correctly fix the subset of Defects4J bugs that can be accurately localized.
> kPAR: A baseline for the research community

kPAR: Java implementation of PAR (Kim et al. ICSE 2013) + Gzoltar-0.1.1 + Ochiai.

**Normal FL:** It relies on the ranked list of suspicious code locations reported by a given FL tool.

**File Assumption:** It assumes that the faulty code files are known.

**Method Assumption:** It assumes that the faulty methods are known.

**Line Assumption:** It assumes that the faulty code lines are known. No fault localization is then used.
kPAR: comparison

Number of Defects4J bugs fixed by kPAR with four FL configurations.

<table>
<thead>
<tr>
<th>FL Conf.</th>
<th>Chart (C)</th>
<th>Closure (Cl)</th>
<th>Lang (L)</th>
<th>Math (M)</th>
<th>Mockito (Moc)</th>
<th>Time (T)</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal FL</td>
<td>3/10</td>
<td>5/9</td>
<td>1/8</td>
<td>7/18</td>
<td>1/2</td>
<td>1/2</td>
<td>18/49</td>
</tr>
<tr>
<td>File Assumption</td>
<td>4/7</td>
<td>6/13</td>
<td>1/8</td>
<td>7/15</td>
<td>2/2</td>
<td>2/3</td>
<td>22/48</td>
</tr>
<tr>
<td>Method Assumption</td>
<td>4/6</td>
<td>7/16</td>
<td>1/7</td>
<td>7/15</td>
<td>2/2</td>
<td>2/3</td>
<td>23/49</td>
</tr>
<tr>
<td>Line Assumption</td>
<td>7/8</td>
<td>11/16</td>
<td>4/9</td>
<td>9/16</td>
<td>2/2</td>
<td>3/4</td>
<td>36/55</td>
</tr>
</tbody>
</table>

With better fault localization results, kPAR can correctly fix more bugs.
What about Test Suites?
Assumption of Complete/Reliable Test suite
A relevant test case reproducing the bug may not be readily available, when a bug report is submitted to the issue tracking system.
How to repair without future information?

Fault Localization

Buggy Program

FL with GZoltar

A ranked list of suspicious code locations

Fix Pattern Selection

Fix pattern database

Selected fix pattern

Code AST

Next suspicious code fragment

Next fix pattern

Fault Localization

Patch Generation

Mutate suspicious code

Patch Candidates

Next patch candidate

Patch Validation

Pass

Fail

Test Cases
iFixR: Bug Report driven Program Repair

Step 0: Distribute to Regions

Step 1: Standard IRBL
...
...
...
...
...

Step 2: IR-based fault localization

Step 3: Fix pattern matching

Step 4: Mutation of suspicious code

Step 5: Regression testing

Fault Localization
Fix Pattern Matching
Patch Generation
Patch Validation

Buggy Program
Suspicious Code Locations
Fix patterns
Code Elements (AST)
Patch Candidates
Manual Validation
Developer Test

Select fix pattern

Mutate suspicious code

Fix patterns

Regression Testing

Manual Validation

Fix patterns

Mutate suspicious code
iFixR - Fault Localization

Statement level Information Retrieval Fault Localization (IRFL)

- Extract Text
- Extract Statements
- Feature Vectors
- Similarity
- Suspicious Statements with weight scores
- F_score
- X
- S_score

Suspicious Code Files

Bug Report

A set of code files

Model

Suspicious Code Files

37
# iFixR - Fix Pattern-based Patch Generation

All fix patterns in the APR community

<table>
<thead>
<tr>
<th>Pattern Description</th>
<th>Used by</th>
<th>Pattern Description</th>
<th>Used by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert Cast Checker</td>
<td>Genesis</td>
<td>Mutate Literal Expression</td>
<td>SimFix</td>
</tr>
<tr>
<td>Insert Null Pointer Checker</td>
<td>NPEFix</td>
<td>Mutate Method Invocation</td>
<td>ELIXIR</td>
</tr>
<tr>
<td>Insert Range Checker</td>
<td>SOFix</td>
<td>Mutate Operator</td>
<td>jMutRepair</td>
</tr>
<tr>
<td>Insert Missed Statement</td>
<td>HDRepair</td>
<td>Mutate Return Statement</td>
<td>SketchFix</td>
</tr>
<tr>
<td>Mutate Conditional Expression</td>
<td>ssFix</td>
<td>Mutate Variable</td>
<td>CapGen</td>
</tr>
<tr>
<td>Mutate Data Type</td>
<td>AVATAR</td>
<td>Move Statement(s)</td>
<td>PAR</td>
</tr>
<tr>
<td>Remove Statement(s)</td>
<td>FixMiner</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
  if (exp instanceof T) {
      ...(T) exp...; ....
  }
```

“Insert Cast Checker” fix pattern
iFixR - Patch Validation

A **patch ordering strategy** to recommend patches with priority

Heuristics to **re-prioritize** the patch candidates

1. Minimal changes
2. Fault localization suspiciousness
3. Affected code elements
Research Questions

**RQ1:** [Fault localization] : To what extent does IR-based fault localization provide reliable results for an APR scenario?

**RQ2:** [Overfitting] : To what extent does IR-based fault localization point to locations that are less subject to overfitting?

**RQ3:** [Patch ordering] : What is the effectiveness of MIMIC’s patch ordering strategy?
Table 5: Fault localization results: IRFL (IR-based) vs. SFL (Spectrum-based) on Defects4J (Math and Lang) bugs.

<table>
<thead>
<tr>
<th></th>
<th>(171 bugs)</th>
<th>Top-1</th>
<th>Top-10</th>
<th>Top-50</th>
<th>Top-100</th>
<th>Top-200</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRFL</td>
<td></td>
<td>25</td>
<td>72</td>
<td>102</td>
<td>117</td>
<td>121</td>
<td>139</td>
</tr>
<tr>
<td>SFL</td>
<td>GZ_v1</td>
<td>26</td>
<td>75</td>
<td>106</td>
<td>110</td>
<td>114</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td>GZ_v2</td>
<td>23</td>
<td>79</td>
<td>119</td>
<td>135</td>
<td>150</td>
<td>156</td>
</tr>
</tbody>
</table>

† GZ\_v1 and GZ\_v2 refer to GZoltar 0.1.1 and 1.6.0 respectively, which are widely used in APR systems for Java programs.

Fine-grained IR-based Fault Localization (IRFL) can be as accurate as Spectrum-based fault localization + it does not require test cases
Overfitting

IRFL vs. SFL impacts on the number of generated genuine/plausible patches for Defects4J bugs.

<table>
<thead>
<tr>
<th></th>
<th>Lang</th>
<th>Math</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRFL Top-1</td>
<td>1/4</td>
<td>3/4</td>
<td>4/8</td>
</tr>
<tr>
<td>SFL Top-1</td>
<td>1/4</td>
<td>6/8</td>
<td>7/12</td>
</tr>
<tr>
<td>IRFL Top-5</td>
<td>3/6</td>
<td>7/14</td>
<td>10/20</td>
</tr>
<tr>
<td>SFL Top-5</td>
<td>2/7</td>
<td>11/17</td>
<td>13/24</td>
</tr>
<tr>
<td>IRFL Top-10</td>
<td>4/9</td>
<td>9/17</td>
<td>13/26</td>
</tr>
<tr>
<td>SFL Top-10</td>
<td>4/11</td>
<td>16/27</td>
<td>20/38</td>
</tr>
<tr>
<td>IRFL Top-20</td>
<td>7/12</td>
<td>9/18</td>
<td>16/30</td>
</tr>
<tr>
<td>SFL Top-20</td>
<td>4/11</td>
<td>18/30</td>
<td>22/41</td>
</tr>
<tr>
<td>IRFL Top-50</td>
<td>7/15</td>
<td>10/22</td>
<td>17/37</td>
</tr>
<tr>
<td>SFL Top-50</td>
<td>4/13</td>
<td>19/34</td>
<td>23/47</td>
</tr>
<tr>
<td>IRFL Top-100</td>
<td>8/18</td>
<td>10/23</td>
<td>18/41</td>
</tr>
<tr>
<td>SFL Top-100</td>
<td>5/14</td>
<td>19/36</td>
<td>24/50</td>
</tr>
<tr>
<td>IRFL All</td>
<td>11/19</td>
<td>10/25</td>
<td>21/44</td>
</tr>
<tr>
<td>SFL All</td>
<td>5/14</td>
<td>19/36</td>
<td>24/50</td>
</tr>
</tbody>
</table>

Dissection of reasons why patches are plausible* but not genuine.

<table>
<thead>
<tr>
<th></th>
<th>Localization Error</th>
<th>Pattern Prioritization</th>
<th>Lack of Fix ingredients</th>
</tr>
</thead>
<tbody>
<tr>
<td>w/ IRFL</td>
<td>6</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>w/ SFL</td>
<td>15</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

*A plausible patch passes all test cases, but may not be semantically equivalent to developer patch (i.e., genuine). We consider a plausible patch to be overfitted to the test suite.

IR-based fault localization lead less to overfitted patches than the code locations suggested by Spectrum-based fault localization.
### Patch Ordering

#### Table 9: Overall performance of iFixR for patch recommendation on the Defects4J benchmark.

<table>
<thead>
<tr>
<th>Recommendation rank</th>
<th>Top-1</th>
<th>Top-5</th>
<th>Top-10</th>
<th>Top-20</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>without</strong> patch re-prioritization</td>
<td>3/3</td>
<td>4/5</td>
<td>6/10</td>
<td>6/10</td>
<td>13/27</td>
</tr>
<tr>
<td><strong>with</strong> patch re-prioritization</td>
<td>3/4</td>
<td>8/13</td>
<td>9/14</td>
<td>10/15</td>
<td>13/27</td>
</tr>
</tbody>
</table>

* x/y: x is the number of bugs for which a *correct* patch is generated; y is the number of bugs for which a *plausible* patch is generated.

Ordering works!
iFixR vs the State-of-the-Art

Table 10: iFixR vs state-of-the-art APR tools.

<table>
<thead>
<tr>
<th>APR tool</th>
<th>Lang*</th>
<th>Math*</th>
<th>Total*</th>
</tr>
</thead>
<tbody>
<tr>
<td>jGenProg [58]</td>
<td>0/0</td>
<td>5/18</td>
<td>5/18</td>
</tr>
<tr>
<td>jKali [58]</td>
<td>0/0</td>
<td>1/14</td>
<td>1/14</td>
</tr>
<tr>
<td>jMutRepair [58]</td>
<td>0/1</td>
<td>2/11</td>
<td>2/12</td>
</tr>
<tr>
<td>HDRepair [35]</td>
<td>2/6</td>
<td>4/7</td>
<td>6/13</td>
</tr>
<tr>
<td>Nopol [92]</td>
<td>3/7</td>
<td>1/21</td>
<td>4/28</td>
</tr>
<tr>
<td>ACS [91]</td>
<td>3/4</td>
<td>12/16</td>
<td>15/20</td>
</tr>
<tr>
<td>ELIXIR [72]</td>
<td>8/12</td>
<td>12/19</td>
<td>20/31</td>
</tr>
<tr>
<td>JAID [12]</td>
<td>1/8</td>
<td>1/8</td>
<td>2/16</td>
</tr>
<tr>
<td>ssFix [89]</td>
<td>5/12</td>
<td>10/26</td>
<td>15/38</td>
</tr>
<tr>
<td>CapGen [83]</td>
<td>5/5</td>
<td>12/16</td>
<td>17/21</td>
</tr>
<tr>
<td>SketchFix [18]</td>
<td>3/4</td>
<td>7/8</td>
<td>10/12</td>
</tr>
<tr>
<td>FixMiner [30]</td>
<td>2/3</td>
<td>12/14</td>
<td>14/17</td>
</tr>
<tr>
<td>LSRepair [43]</td>
<td>8/14</td>
<td>7/14</td>
<td>15/28</td>
</tr>
<tr>
<td>SimFix [19]</td>
<td>9/13</td>
<td>14/26</td>
<td>23/39</td>
</tr>
<tr>
<td>kPAR [47]</td>
<td>1/8</td>
<td>7/18</td>
<td>8/26</td>
</tr>
<tr>
<td>AVATAR [48]</td>
<td>5/11</td>
<td>6/13</td>
<td>11/24</td>
</tr>
<tr>
<td>iFixR_opt</td>
<td>11/19</td>
<td>10/25</td>
<td>21/44</td>
</tr>
<tr>
<td>iFixR_all</td>
<td>6/11</td>
<td>7/16</td>
<td>13/27</td>
</tr>
<tr>
<td>iFixR_top5</td>
<td>3/7</td>
<td>5/6</td>
<td>8/13</td>
</tr>
</tbody>
</table>

- reasonable performance in patch recommendation @Top5 (we assume not having relevant test cases to validate the patch candidates).

- Comparable performance to many state-of-the-art test-based APR tools in the literature.

*x/y: x is the number of bugs for which a correct patch is generated; y is the number of bugs for which a plausible patch is generated.

iFixR_opt: the version of iFixR where available test cases are relevant to the bugs.
iFixR_all: all recommended patches are considered.
iFixR_top5: only top 5 recommended patches are considered.
One Last Thing...

Buggy code can be fixed by simply replacing it with « semantically » similar code...

⇒ Effective for 21 Defects4J Bugs
Is patch generation efficient?
“Time” is not a good metric for efficiency of APR

Distribution CPU times for compiling and testing Defects4J programs

- Machine 1 runs OS X El Capitan 10.11.6 with 2.5 GHz Intel Core i7, 16GB 1600MHz DDR3 RAM.
- Machine 2 runs macOS Mojave 10.14.1 with 2.9 GHz Intel Core i9, 32 GB 2400MHz DDR4 RAM.
“NPC”: Number of Patch Candidates

Efficiency is not yet a widely-valued performance target

The more templates an APR system considers, the more nonsensical and in-plausible patches it will generate.
Can we predict patch correctness?
Representation learning of code changes

- Static feature learning from patches with BERT, Doc2Vec, Code2Vec and CC2Vec
Representation learning of code changes

- Fixed code is "similar" to buggy code!
Cosine similarity as a filter

<table>
<thead>
<tr>
<th>Dataset</th>
<th># CP</th>
<th># IP</th>
<th>Threshold</th>
<th>BERT</th>
<th>CC2Vec</th>
<th>Doc2Vec</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+CP</td>
<td>-IP</td>
<td>+Recall</td>
</tr>
<tr>
<td>Bears, Bugs.jar and Defects4J</td>
<td>893</td>
<td>61,932</td>
<td>1st Qu.</td>
<td>57</td>
<td>48,846</td>
<td>6.4%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Mean</td>
<td>49</td>
<td>51,783</td>
<td>5.5%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QuixBugs</td>
<td>7</td>
<td>1,461</td>
<td>1st Qu.</td>
<td>4</td>
<td>1,387</td>
<td>57.1%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Mean</td>
<td>4</td>
<td>1,378</td>
<td>57.1%</td>
</tr>
</tbody>
</table>

**"# CP" and "# IP" stand for the number of correct and incorrect patches, respectively. "# +CP" means the number of correct patches that can be ranked upon the threshold, while "# -IP" means the number of incorrect patches that can be filtered out by the threshold. "+Recall" and "-Recall" represent the recall of identifying correct patches and filtering out incorrect patches, respectively.**

Similarity thresholds can be used to filter out some incorrect patches!
Learning to classify patches

Embeddings offer reasonable performance for **statically** predicting patch correctness!
Next steps
New contexts/criteria

- Based on User Input
- Correctness standard
- Repair
- With feedback loop
- With explanations
ERC Starting grant – « NATURAL Program Repair »

1. Bug report submitted by software user.
2. Feedback from human expert contributors.
3. Repair Bot processes the feedback to prepare a potential release candidate.
4. Pull request is prepared with patch, explanations, examples, and test cases.
5. Pull request is merged into mainline.

Potential release candidate

Pull Request

ok with the pull request?

yes

no

maintainer

pull request is rejected
Merci!