Automated Program Repair

Opportunities, Challenges, Advances

Chris Timperley

Carnegie Mellon University

isr institute for SOFTWARE RESEARCH

About me...

ctimperley@cmu.edu

Research Interests:

- Automated Program Repair
- Fault Localisation
- GI for Robotics
- Automated Test Generation

Recent Projects:

- **BugZoo:** reproducible studies of historical bugs
- **Rooibos:** language-independent, syntax-aware search and transformation.
- **Houston*:*** automated testing for robotics

Postdoc
Carnegie Mellon University, USA
w/ Claire Le Goues

Carnegie Mellon University

MEng (2013), PhD (2017)
University of York, UK
w/ *Susan Stepney*

UNIVERSITY of York

Purpose of Talk:

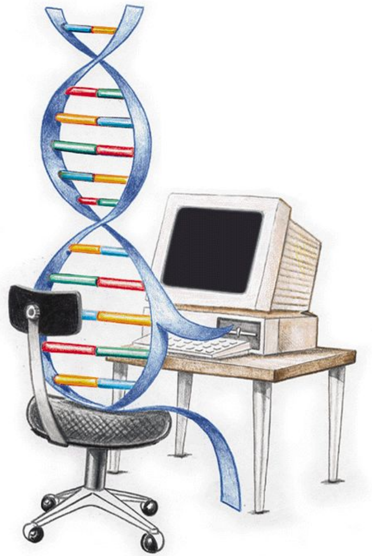
Challenge existing views, and identify opportunities.

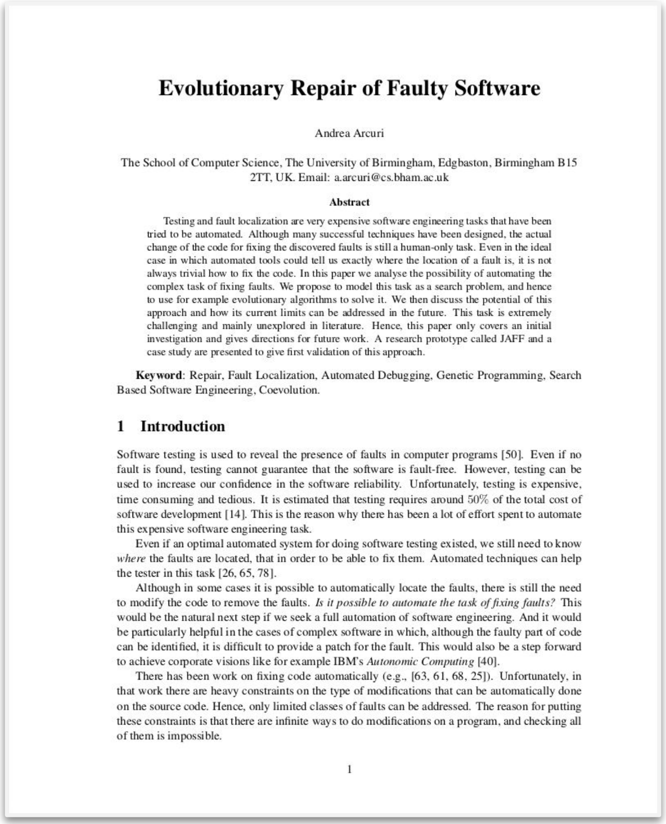

SILVER
SHEFFIELD.

2008

Genetic Programming to modify existing programs, rather than building them from scratch.

Demonstrates concept of automated program repair.





Evolutionary Repair of Faulty Software

Andrea Arcuri

The School of Computer Science, The University of Birmingham, Edgbaston, Birmingham B15 2TT, UK. Email: a.arcuri@cs.bham.ac.uk

Abstract

Testing and fault localization are very expensive software engineering tasks that have been tried to be automated. Although many successful techniques have been designed, the actual change of the code for fixing the discovered faults is still a human-only task. Even in the ideal case in which automated tools could tell us exactly where the location of a fault is, it is not always trivial how to fix the code. In this paper we analyse the possibility of automating the complex task of fixing faults. We propose to model this task as a search problem, and hence to use for example evolutionary algorithms to solve it. We then discuss the potential of this approach and how its current limits can be addressed in the future. This task is extremely challenging and mainly unexplored in literature. Hence, this paper only covers an initial investigation and gives directions for future work. A research prototype called JAFF and a case study are presented to give first validation of this approach.

Keyword: Repair, Fault Localization, Automated Debugging, Genetic Programming, Search Based Software Engineering, Coevolution.

1 Introduction

Software testing is used to reveal the presence of faults in computer programs [50]. Even if no fault is found, testing cannot guarantee that the software is fault-free. However, testing can be used to increase our confidence in the software reliability. Unfortunately, testing is expensive, time consuming and tedious. It is estimated that testing requires around 50% of the total cost of software development [14]. This is the reason why there has been a lot of effort spent to automate this expensive software engineering task.

Even if an optimal automated system for doing software testing existed, we still need to know *where* the faults are located, that in order to be able to fix them. Automated techniques can help the tester in this task [26, 65, 78].

Although in some cases it is possible to automatically locate the faults, there is still the need to modify the code to remove the faults. *Is it possible to automate the task of fixing faults?* This would be the natural next step if we seek a full automation of software engineering. And it would be particularly helpful in the cases of complex software in which, although the faulty part of code can be identified, it is difficult to provide a patch for the fault. This would also be a step forward to achieve corporate visions like for example IBM's *Autonomic Computing* [40].

There has been work on fixing code automatically (e.g., [63, 61, 68, 25]). Unfortunately, in that work there are heavy constraints on the type of modifications that can be automatically done on the source code. Hence, only limited classes of faults can be addressed. The reason for putting these constraints is that there are infinite ways to do modifications on a program, and checking all of them is impossible.

1

Evolutionary repair of faulty software.
Andrea Arcuri. 2011.
Applied Soft Computing 11, 4 (June 2011), 3494-3514.

GenProg

HOME PEOPLE EXPERIMENTS PUBLICATIONS VIDEOS REPOSITORY

GenProg

Evolutionary Program Repair

The Problem

Software engineering is expensive, summing to over one half
of the US GDP annually. Software maintenance accounts for o
of that life cycle cost, and a key aspect of maintenance is fixin
existing programs. Unfortunately, the number of reported bu
available development resources. It is common for a popular p
hundreds of new bug reports filed every day.

Software maintenance is expensive. GenProg reduces softwa
maintenance costs by automatically producing patches (repai
defects.

Our Approach

Many bugs can be fixed with just a few changes to a program's
Human repairs often involve inserting new code and deleting
existing code. GenProg uses those same building blocks to sea
automatically.

GenProg uses *genetic programming* to search for repairs. Ou

Automatically Finding Patches Using Genetic Programming *

Westley Weimer, University of Virginia, weimer@virginia.edu
ThanhVu Nguyen, University of New Mexico, tnguyen@cs.unm.edu
Claire Le Goues, University of Virginia, legoues@virginia.edu
Stephanie Forrest, University of New Mexico, forrest@cs.unm.edu

Abstract

Automatic program repair has been a longstanding goal in software engineering, yet debugging remains a largely manual process. We introduce a fully automated method for locating and repairing bugs in software. The approach works on off-the-shelf legacy applications and does not require formal specifications, program annotations or special coding practices. Once a program fault is discovered, an extended form of genetic programming is used to evolve program variants until one is found that both retains required functionality and also avoids the defect in question. Standard test cases are used to exercise the fault and to encode program requirements. After a successful repair has been discovered, it is minimized using structural differencing algorithms and delta debugging. We describe the proposed method and report experimental results demonstrating that it can successfully repair ten different C programs totaling 63,000 lines in under 200 seconds, on average.

1 Introduction

Fixing bugs is a difficult, time-consuming, and manual process. Some reports place software maintenance, traditionally defined as any modification made on a system after its delivery, at 90% of the total cost of a typical software project [27]. Modifying existing code, repairing defects, and otherwise evolving software are major parts of those costs [24]. The number of outstanding software defects typically exceeds the resources available to address them [4]. Mature software projects are forced to ship with both known and unknown bugs [21] because they lack the development resources to deal with every defect. For example, in 2005, one Mozilla developer claimed that, "everyday, almost 300 bugs appear [. . .] far too much for only the Mozilla programmers to handle" [5, p. 363].

To alleviate this burden, we propose an automatic technique for repairing program defects. Our approach does not require difficult formal specifications, program annotations or special coding practices. Instead, it works on off-the-shelf legacy applications and readily-available testcases. We use genetic programming to evolve program variants until one is found that both retains required functionality and also avoids the defect in question. Our technique takes as input a program, a set of successful positive testcases that encode required program behavior, and a failing negative testcase that demonstrates a defect.

Genetic programming (GP) is a computational method inspired by biological evolution, which discovers computer programs tailored to a particular task [19]. GP maintains a population of individual programs. Computational analogs of biological mutation and crossover produce program variants. Each variant's suitability is evaluated using a user-defined fitness function, and successful variants are selected for continued evolution. GP has solved an impressive range of problems (e.g., see [1]), but to our knowledge it has not been used to evolve off-the-shelf legacy software.

A significant impediment for an evolutionary algorithm like GP is the potentially infinite-size search space it must sample to find a correct program. To address this problem, we introduce two key innovations. First, we restrict the algorithm to only produce changes that are based on structures in other parts of the program. In essence, we hypothesize that a program that is missing important functionality (e.g., a null check) will be able to copy and adapt it from another location in the program. Second, we constrain the genetic operations of mutation and crossover to operate only on the region of the program that is relevant to the error (that is, the portions of the program that were on the execution path that produced the error). Combining these insights, we demonstrate automatically generated repairs for ten C programs totaling 63,000 lines of code.

We use GP to maintain a population of variants of that program. Each variant is represented as an abstract syntax tree (AST) paired with a weighted program path. We modify variants using two genetic algorithm operations, crossover and mutation, specifically targeted to this repre-

*This research was supported in part by National Science Foundation Grants CNS 0627523 and CNS 0716478, Air Force Office of Scientific Research grant FA9550-07-1-0532, as well as gifts from Microsoft Research. No official endorsement should be inferred.

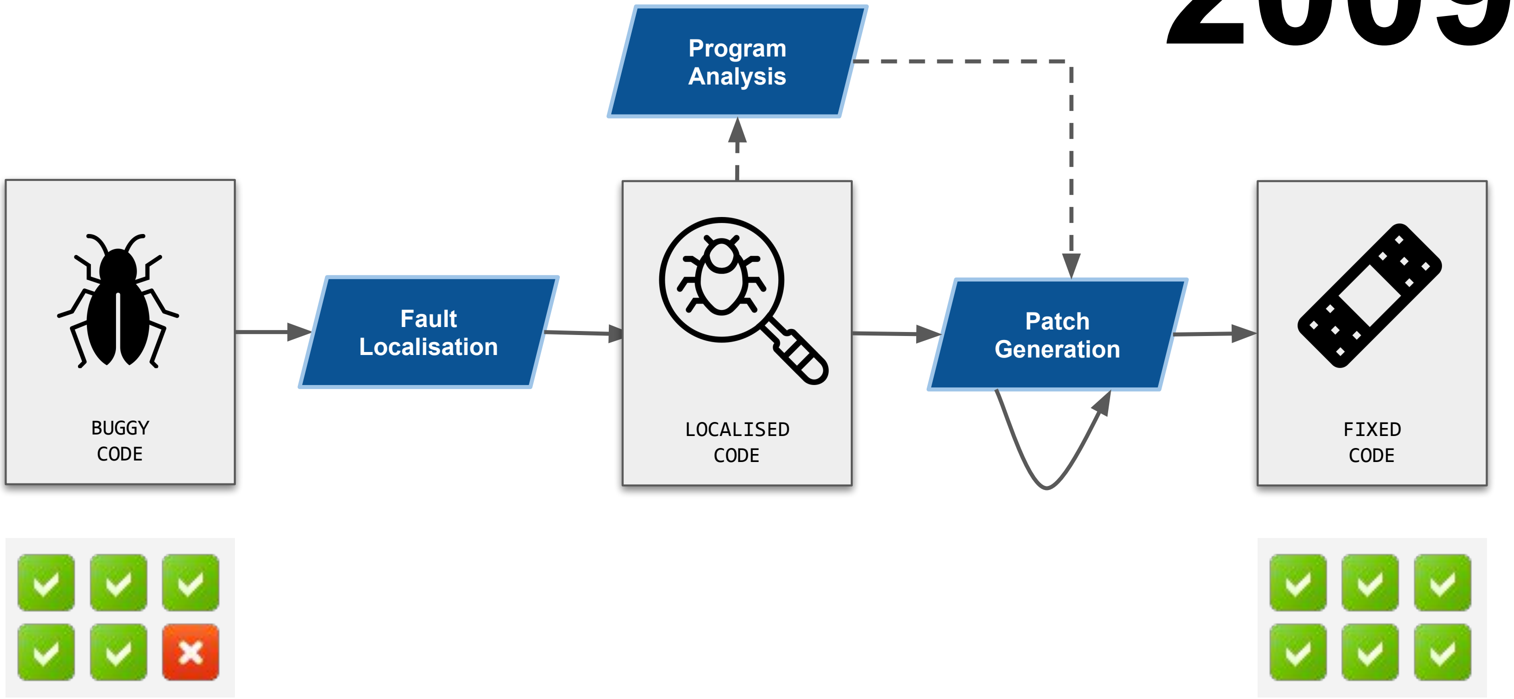
ICSE'09, May 16–24, 2009, Vancouver, Canada
978-1-4244-3452-7/09/$25.00 © 2009 IEEE 364

2009

GenProg demonstrates
program repair on
real-world C programs

Automatically Finding Patches Using Genetic Programming.
Westley Weimer, ThanVu Nguyen, Claire Le Goues, Stephanie Forrest.
Proceedings of International Conference on Software Engineering. ICSE '09. 2009.

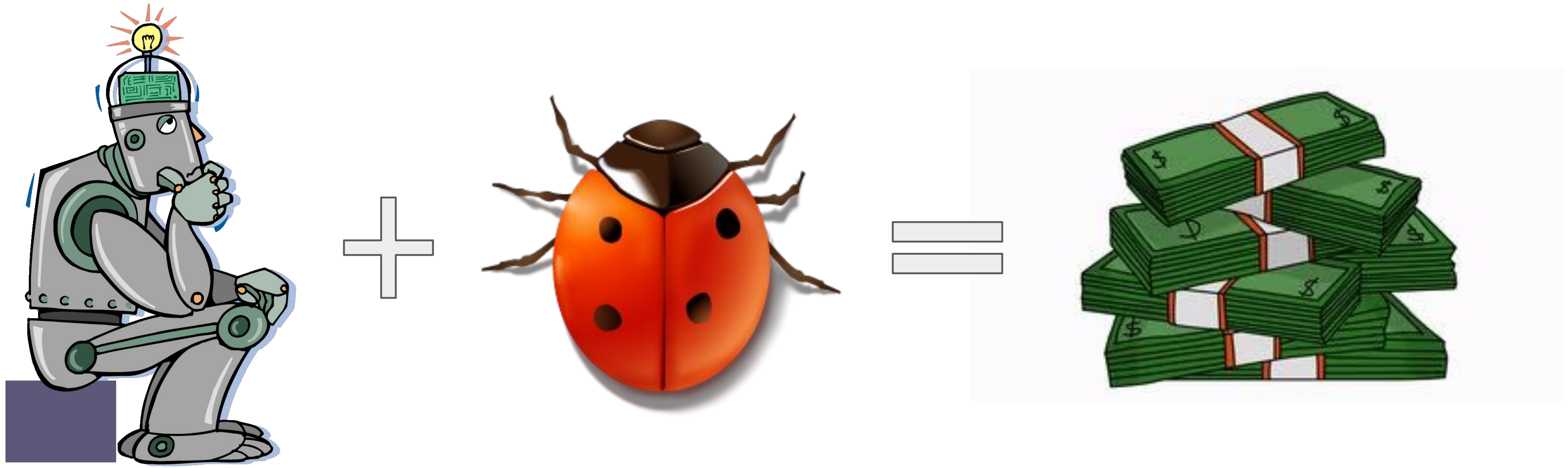
2009–



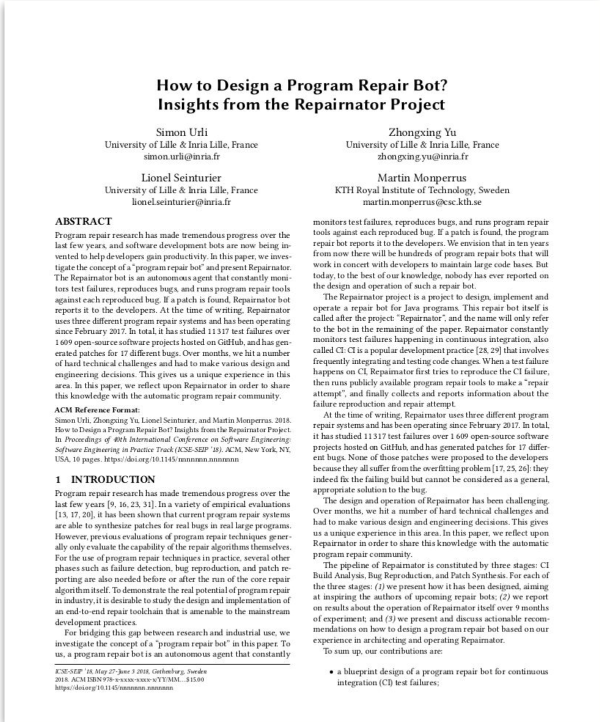
Program Analysis
BUGGY CODE
Fault Localisation
LOCALISED CODE
Patch Generation
FIXED CODE

Where are the program repair bots?

Or, why aren't we all filthy rich yet?



ICSE SEIP ‘18



How to Design a Program Repair Bot? Insights from the Repairnator Project

Simon Urli
University of Lille & Inria Lille, France
simon.urli@inria.fr

Zhongxing Yu
University of Lille & Inria Lille, France
zhongxing.yu@inria.fr

Lionel Seinturier
University of Lille & Inria Lille, France
lionel.seinturier@inria.fr

Martin Monperrus
KTH Royal Institute of Technology, Sweden
martin.monperrus@csc.kth.se

ABSTRACT

Program repair research has made tremendous progress over the last few years, and software development bots are now being invented to help developers gain productivity. In this paper, we investigate the concept of a "program repair bot" and present Repairnator. The Repairnator bot is an autonomous agent that constantly monitors test failures, reproduces bugs, and runs program repair tools against each reproduced bug. If a patch is found, Repairnator bot reports it to the developers. At the time of writing, Repairnator uses three different program repair systems and has been operating since February 2017. In total, it has studied 11 317 test failures over 1 609 open-source software projects hosted on GitHub, and has generated patches for 17 different bugs. Over months, we hit a number of hard technical challenges and had to make various design and engineering decisions. This gives us a unique experience in this area. In this paper, we reflect upon Repairnator in order to share this knowledge with the automatic program repair community.

ACM Reference Format:
Simon Urli, Zhongxing Yu, Lionel Seinturier, and Martin Monperrus. 2018. How to Design a Program Repair Bot? Insights from the Repairnator Project. In *Proceedings of 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP '18)*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Program repair research has made tremendous progress over the last few years [9, 16, 23, 31]. In a variety of empirical evaluations [13, 17, 20], it has been shown that current program repair systems are able to synthesize patches for real bugs in real large programs. However, previous evaluations of program repair techniques generally only evaluate the capability of the repair algorithms themselves. For the use of program repair techniques in practice, several other phases such as failure detection, bug reproduction, and patch reporting are also needed before or after the run of the core repair algorithm itself. To demonstrate the real potential of program repair in industry, it is desirable to study the design and implementation of an end-to-end repair toolchain that is amenable to the mainstream development practices.

For bridging this gap between research and industrial use, we investigate the concept of a "program repair bot" in this paper. To us, a program repair bot is an autonomous agent that constantly monitors test failures, reproduces bugs, and runs program repair tools against each reproduced bug. If a patch is found, the program repair bot reports it to the developers. We envision that in ten years from now there will be hundreds of program repair bots that will work in concert with developers to maintain large code bases. But today, to the best of our knowledge, nobody has ever reported on the design and operation of such a repair bot.

The Repairnator project is a project to design, implement and operate a repair bot for Java programs. This repair bot itself is called after the project: "Repairnator", and the name will only refer to the bot in the remaining of the paper. Repairnator constantly monitors test failures happening in continuous integration, also called CI: CI is a popular development practice [28, 29] that involves frequently integrating and testing code changes. When a test failure happens on CI, Repairnator first tries to reproduce the CI failure, then runs publicly available program repair tools to make a "repair attempt", and finally collects and reports information about the failure reproduction and repair attempt.

At the time of writing, Repairnator uses three different program repair systems and has been operating since February 2017. In total, it has studied 11 317 test failures over 1 609 open-source software projects hosted on GitHub, and has generated patches for 17 different bugs. None of those patches were proposed to the developers because they all suffer from the overfitting problem [17, 25, 26]: they indeed fix the failing build but cannot be considered as a general, appropriate solution to the bug.

The design and operation of Repairnator has been challenging. Over months, we hit a number of hard technical challenges and had to make various design and engineering decisions. This gives us a unique experience in this area. In this paper, we reflect upon Repairnator in order to share this knowledge with the automatic program repair community.

The pipeline of Repairnator is constituted by three stages: CI Build Analysis, Bug Reproduction, and Patch Synthesis. For each of the three stages: *(1)* we present how it has been designed, aiming at inspiring the authors of upcoming repair bots; *(2)* we report on results about the operation of Repairnator itself over 9 months of experiment; and *(3)* we present and discuss actionable recommendations on how to design a program repair bot based on our experience in architecting and operating Repairnator.

To sum up, our contributions are:

- a blueprint design of a program repair bot for continuous integration (CI) test failures;

ICSE-SEIP '18, May 27-June 3 2018, Gothenburg, Sweden
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

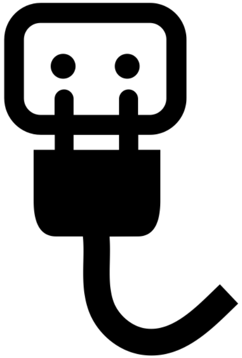
they exist!

(note: we found out about them a few weeks ago)

How to Design a Program Repair Bot? Insights from the Repairnator Project.
Simon Urli, Zhongxing Yu, Lionel Seinturier, and Martin Monperrus.
Proceedings of the International Conference on Software Engineering . ICSE ‘18.

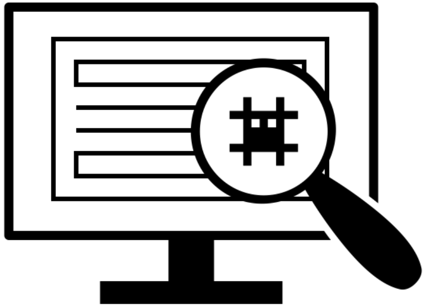
How do we deploy?

What do developers need?

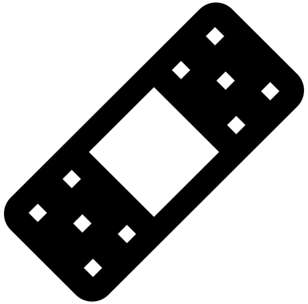


Easy Integration

Timeliness

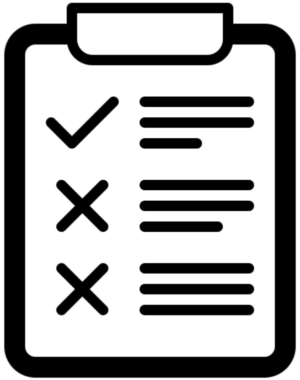


Bug Information



Patches

The big assumption: The existence of tests



requires test cases

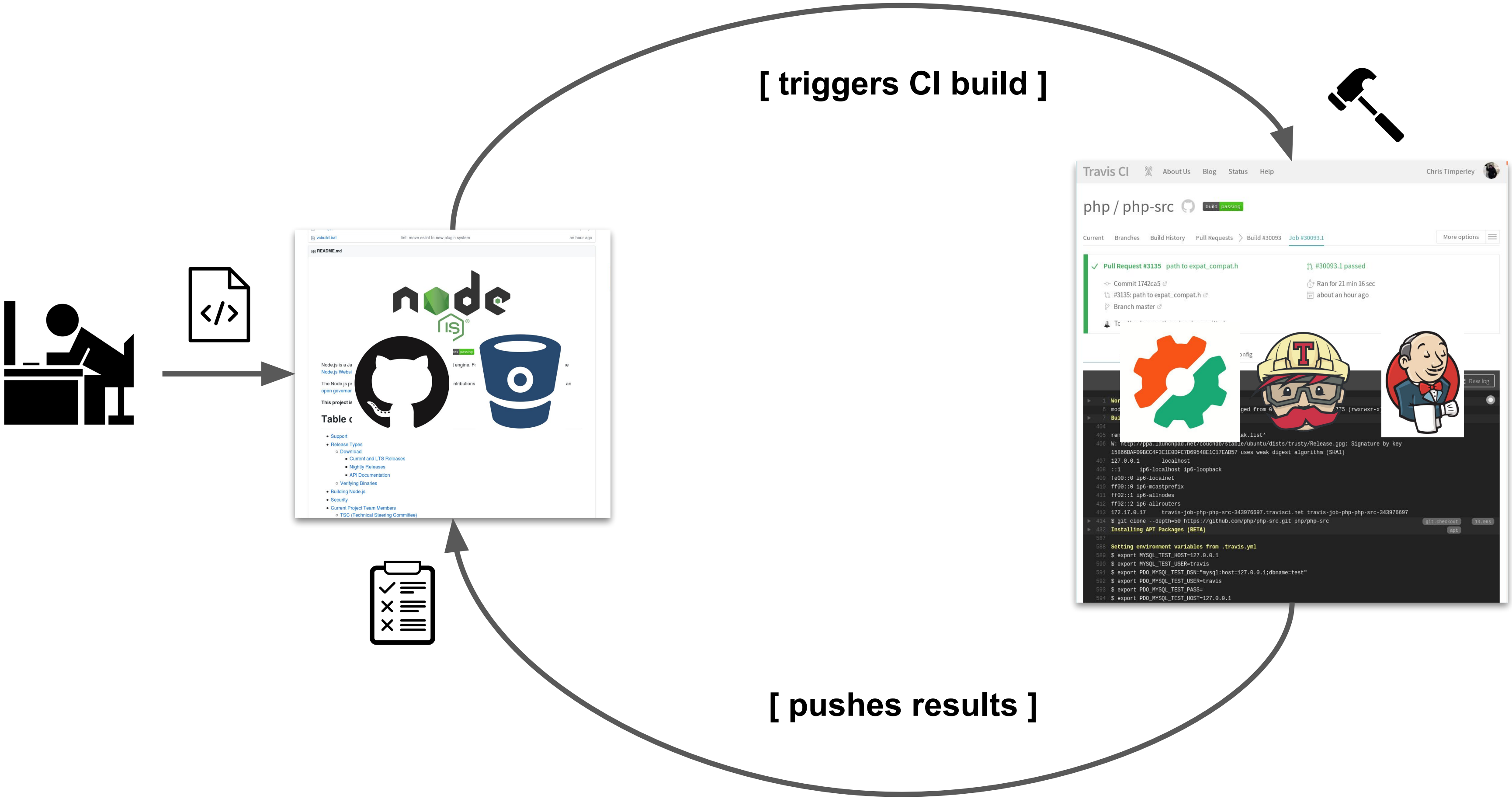
Software engineering has changed since APR was introduced.

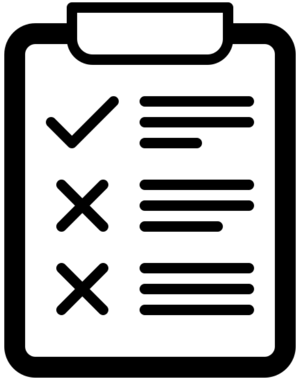
docker

STOMP OUT BUGS!

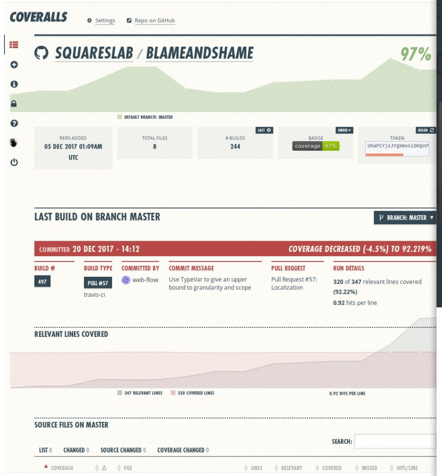
COVERALLS

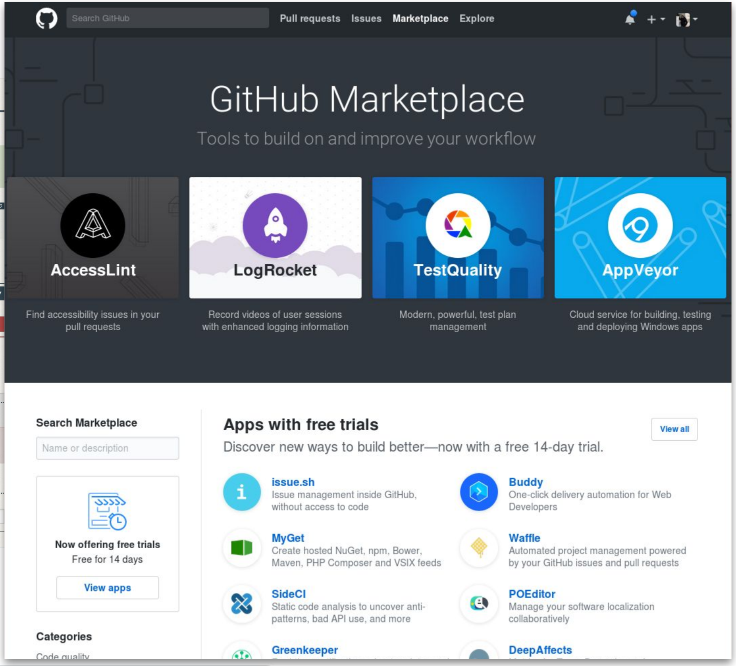
COVERITY

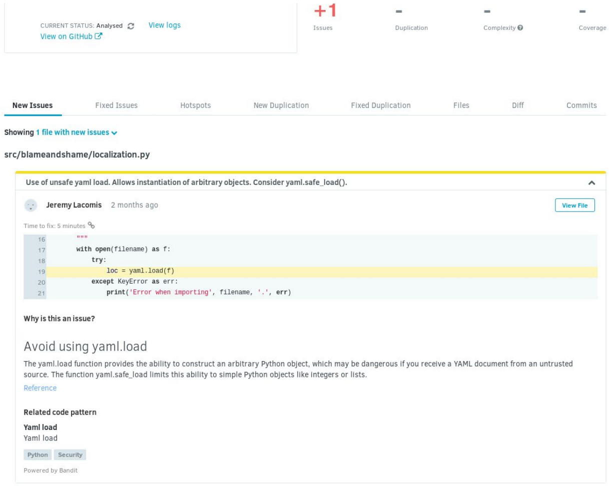

[triggers CI build]
[pushes results]
Travis CI
php / php-src
README.md



requires test cases


COVERALLS
Settings
Repo on GitHub
SQUARESLAB / BLAMEANDSHAME
97%
LAST BUILD ON BRANCH MASTER
COMMITTED 20 DEC 2017 - 14:12
COVERAGE DECREASED (-4.5%) TO 92.219%
RELEVANT LINES COVERED
SOURCE FILES ON MASTER


Search GitHub
Pull requests
Issues
Marketplace
Explore
GitHub Marketplace
Tools to build on and improve your workflow
AccessLint
Find accessibility issues in your pull requests
LogRocket
Record videos of user sessions with enhanced logging information
TestQuality
Modern, powerful, test plan management
AppVeyor
Cloud service for building, testing and deploying Windows apps
Search Marketplace
Name or description
Now offering free trials
Free for 14 days
View apps
Categories
Apps with free trials
View all
Discover new ways to build better—now with a free 14-day trial.
issue.sh
Issue management inside GitHub, without access to code
Buddy
One-click delivery automation for Web Developers
MyGet
Create hosted NuGet, npm, Bower, Maven, PHP Composer and VSIX feeds
Waffle
Automated project management powered by your GitHub issues and pull requests
SideCI
Static code analysis to uncover anti-patterns, bad API use, and more
POEditor
Manage your software localization collaboratively
Greenkeeper
DeepAffects

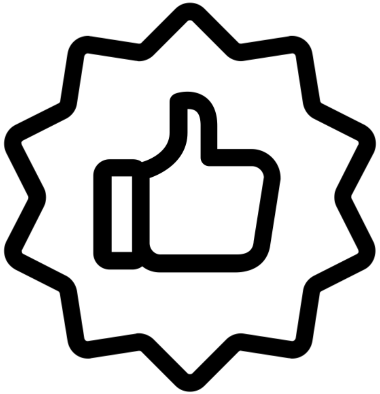

+1
Issues
Duplication
Complexity
Coverage
New Issues
Fixed Issues
Hotspots
New Duplication
Fixed Duplication
Files
Diff
Commits
src/blameandshame/localization.py
Use of unsafe yaml load. Allows instantiation of arbitrary objects. Consider yaml.safe_load().
Jeremy Lacomis
2 months ago
View File
with open(filename) as f:
try:
loc = yaml.load(f)
except KeyError as err:
print('Error when importing', filename, '.', err)
Why is this an issue?
Avoid using yaml.load
The yaml.load function provides the ability to construct an arbitrary Python object, which may be dangerous if you receive a YAML document from an untrusted source. The function yaml.safe_load limits this ability to simple Python objects like integers or lists.
Reference
Related code pattern
Yaml load
Yaml load
Python
Security
Powered by Bandit


code quality
B

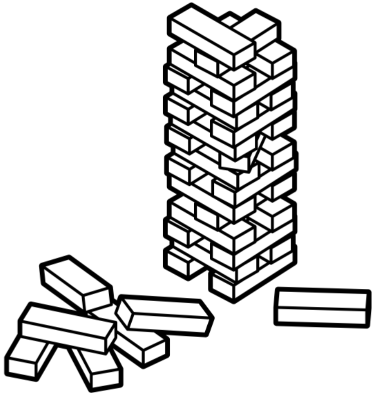

coveralls commented on Dec 8, 2017
coverage
97%
Coverage increased (+0.7%) to 96.774% when pulling fa4a45a on commits-to-function into 95b956e on master.

What are the challenges?

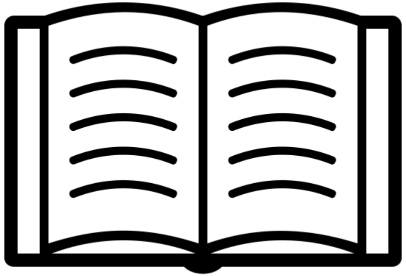
Challenges



Patch Quality



Scalability



Expressiveness

Challenges: Patch Quality

BUGGY

```
max = 0;
if(a > b)
    max  = a;
if(b > a)
    max = b;
```

a	b	max	
3	2	3	✓
4	5	5	✓
7	7	0	✗

CORRECT

```
max = a;
if(a > b)
    max  = a;
if(b > a)
    max = b;
```

a	b	max	
3	2	3	✓
4	5	5	✓
7	7	7	✓

PLAUSIBLE

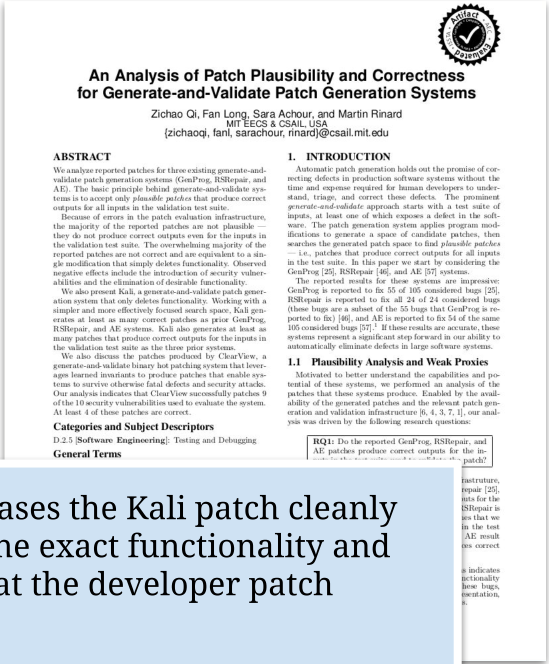
```
max = 7;
if(a > b)
    largest = a;
if(b > a)
    largest = b;
```

a	b	max	
3	2	3	✓
4	5	5	✓
7	7	7	✓

CONTROVERSIAL

Not all good patches are correct.

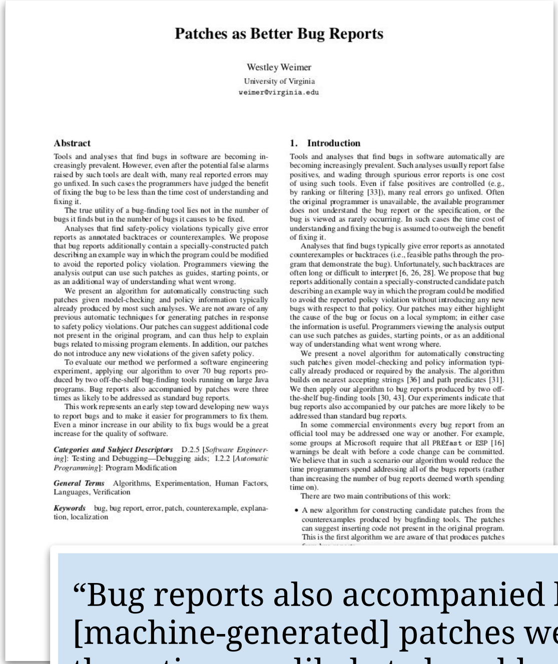
ISSTA '15



"In many cases the Kali patch cleanly identifies the exact functionality and location that the developer patch modifies"

"The Kali and developer patches typically modify common functionality and variables."

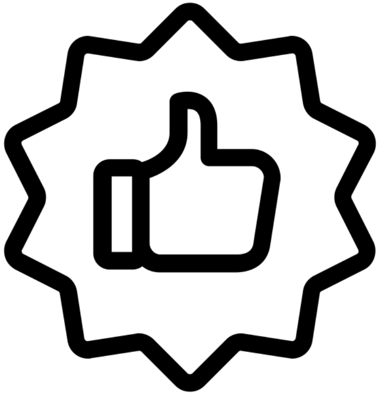
GPCE '06



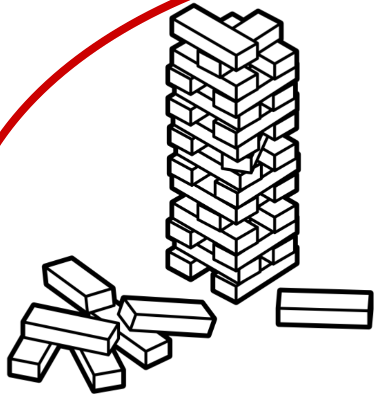
"Bug reports also accompanied by [machine-generated] patches were three times as likely to be addressed as standard bug reports."

Correctness is a major challenge,
but overfitted patches can still be useful.

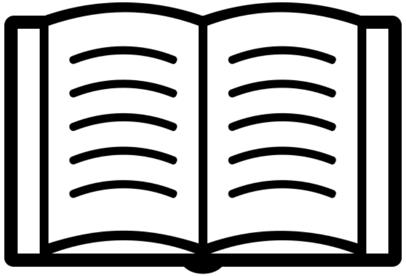
Challenges



Patch Quality

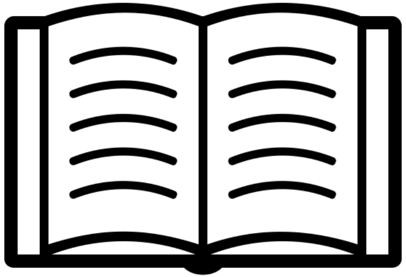


Scalability



Expressiveness

Challenges: Expressiveness vs. Scalability

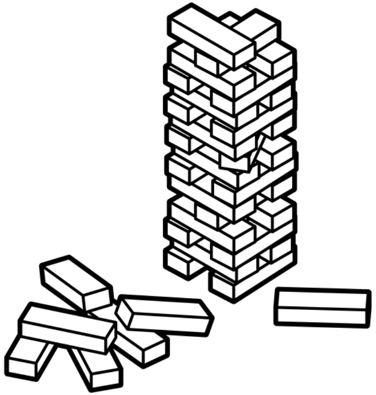


Expressiveness

Allows the program to be changed in a greater number of ways, increasing the odds of finding a modification that produces a repair.

- larger corpus of fix ingredients
- wider set of program transformations
- granular modifications to the program

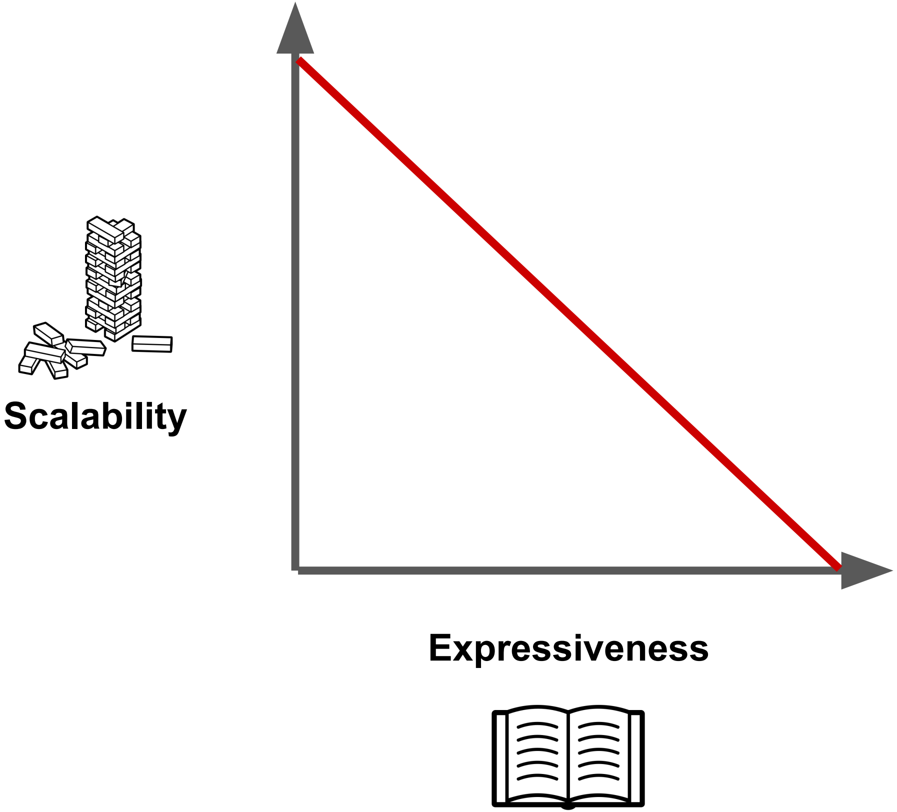
Challenges: Expressiveness vs. Scalability



Scalability

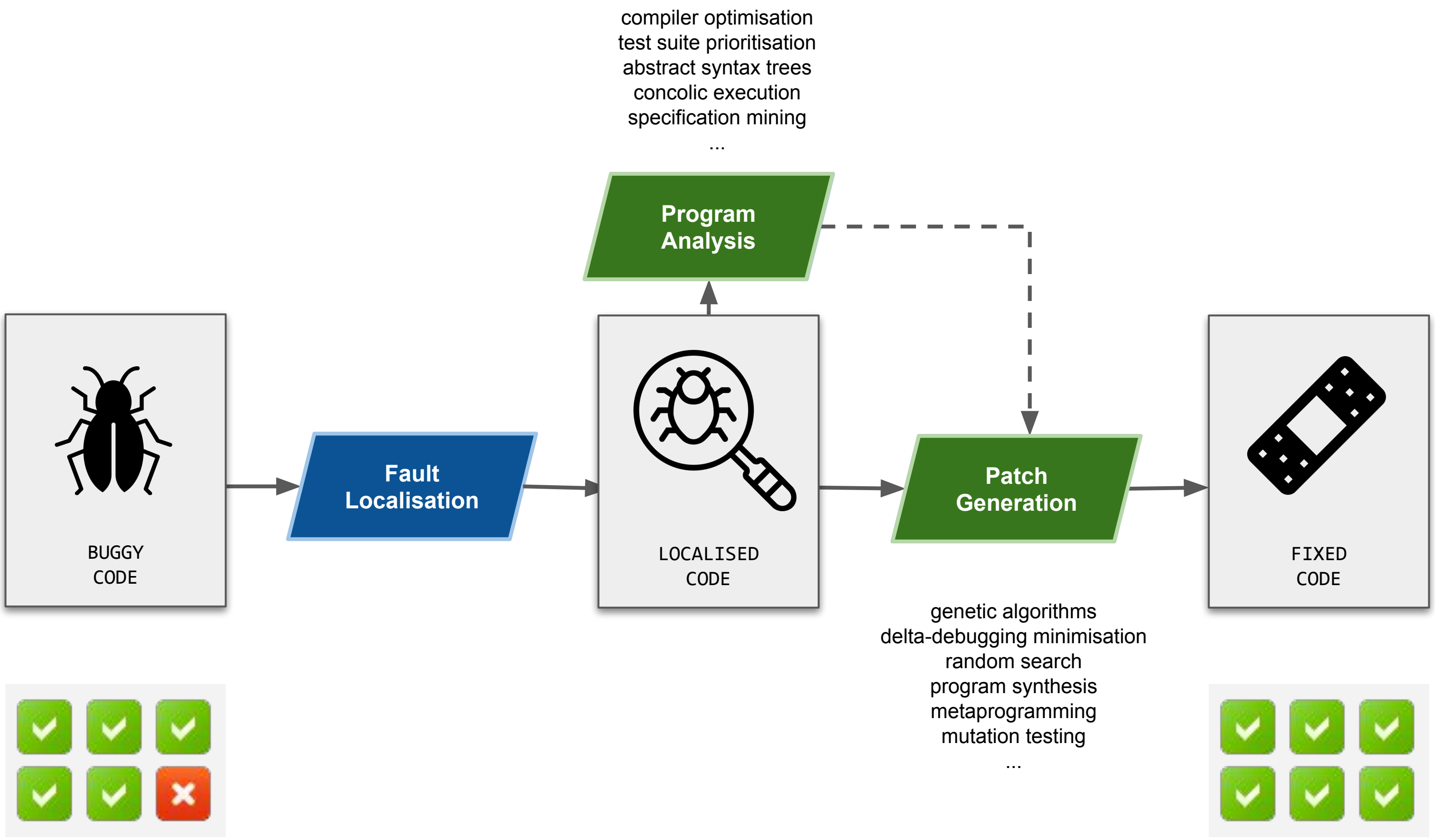
Time taken to discover a patch is a function of:

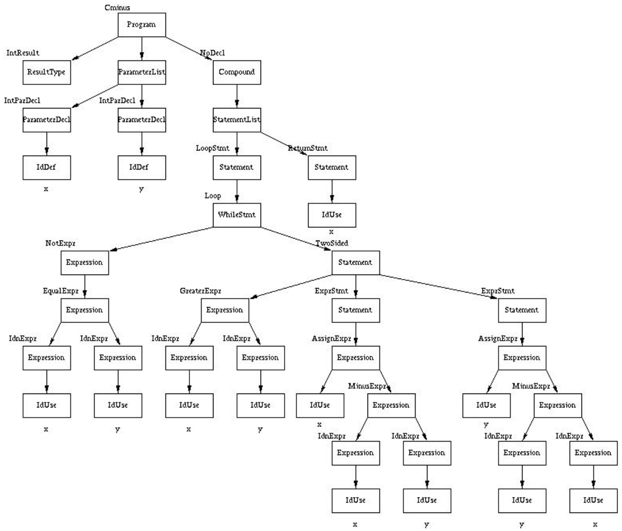
- patch size
- program size
- expressiveness
- ...


Scalability
Expressiveness

How can we make APR both scalable and expressive?

Observation:
APR inherited most of its technologies.


compiler optimisation
test suite prioritisation
abstract syntax trees
concolic execution
specification mining
...
Program Analysis
BUGGY CODE
Fault Localisation
LOCALISED CODE
Patch Generation
FIXED CODE
genetic algorithms
delta-debugging minimisation
random search
program synthesis
metaprogramming
mutation testing
...



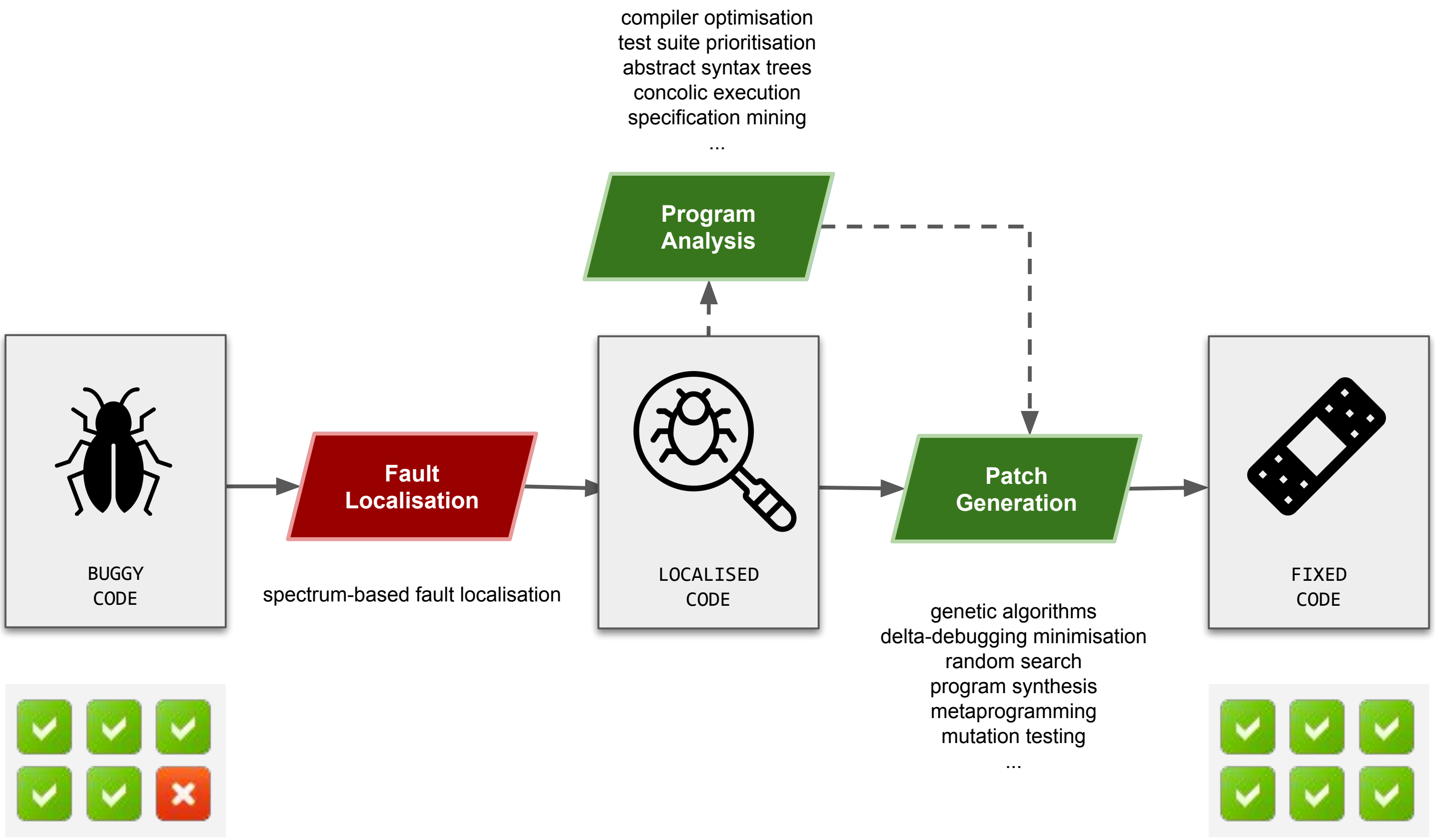
Abstract Syntax Tree

APPEND 12 3

SWAP 5 6

DELETE 4

Patch Representation


compiler optimisation
test suite prioritisation
abstract syntax trees
concolic execution
specification mining
...
Program Analysis
BUGGY CODE
Fault Localisation
spectrum-based fault localisation
LOCALISED CODE
Patch Generation
FIXED CODE
genetic algorithms
delta-debugging minimisation
random search
program synthesis
metaprogramming
mutation testing
...

Everyone is *still* using spectrum-based fault localisation!

- **2017:** ssFix, Repairnator, NOPOL, ...
- **2016:** History-Driven Program Repair, ...
- **2015:** Angelix, SearchRepair, ...
- **2014:** Astor, RSRepair, ...
- **2013:** SemFix, ...
- **2011:** AE, ...
- ...
- **2009:** GenProg

Recap: Spectra-Based Fault Localisation

```c
int year, days;
year = 1980;
days = atoi(argv[1]);
while (days > 365) {
  if (isLeapYear(year)){
    if (days > 366) {
      days -= 366;
      year += 1;
    } else { }
  } else {
    days -= 365;
    year += 1;
  }
}
return year;
```

Passing Test #1

```c
int year, days;
year = 1980;
days = atoi(argv[1]);
while (days > 365) {
  if (isLeapYear(year)){
    if (days > 366) {
      days -= 366;
      year += 1;
    } else { }
  } else {
    days -= 365;
    year += 1;
  }
}
return year;
```

Passing Test #2

```c
int year, days;
year = 1980;
days = atoi(argv[1]);
while (days > 365) {
  if (isLeapYear(year)){
    if (days > 366) {
      days -= 366;
      year += 1;
    } else { }
  } else {
    days -= 365;
    year += 1;
  }
}
return year;
```

Passing Test #3

```c
int year, days;
year = 1980;
days = atoi(argv[1]);
while (days > 365) {
  if (isLeapYear(year)){
    if (days > 366) {
      days -= 366;
      year += 1;
    } else { }
  } else {
    days -= 365;
    year += 1;
  }
}
return year;
```

Failing Test #1

$$\frac{e_f}{e_f + n_f + e_p}$$

Jaccard

$$\frac{e_f}{\sqrt{(e_f + n_f) \cdot (e_f + e_p)}}$$

Ochiai

$$\left| \frac{e_f}{e_f + n_f} - \frac{e_p}{e_p + n_p} \right|$$

AMPLE

$$\begin{cases} 0, & \text{if } \mathrm{e}_f + e_p = 0 \\ 1.0, & \text{if } \mathrm{e}_f > 0 \wedge e_p = 0 \\ 0.1, & \text{otherwise} \end{cases}$$

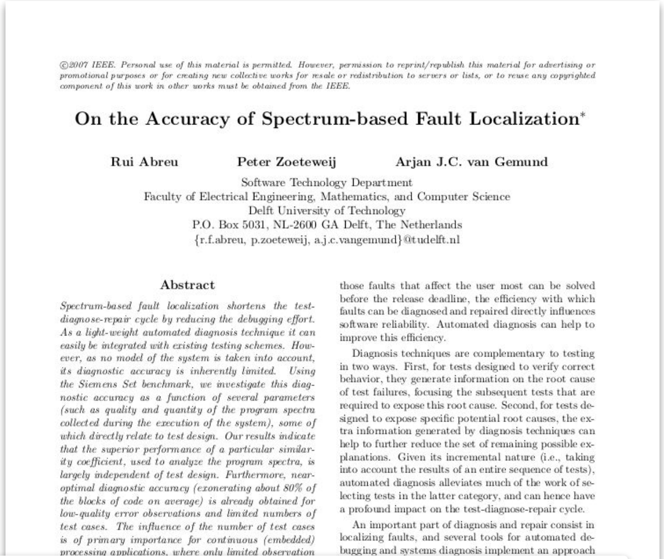
GenProg

$$\frac{\frac{e_f}{e_f+n_f}}{\frac{e_p}{e_p+n_p} + \frac{e_f}{e_f+n_f}}$$

Tarantula

Shin Yoo. 2012. Evolving human competitive spectra-based fault localisation techniques. In Proceedings of the 4th international conference on Search Based Software Engineering (SSBSE'12), Gordon Fraser and Jerffeson Teixeira de Souza (Eds.). Springer-Verlag, Berlin, Heidelberg, 244-258.

```c
if (namelist == NULL) {
    return -1;
}

if (!(dirp = opendir(dirname))) {
    return -1;
}

while (!php_readdir_r(dirp, (struct dirent *)entry, &dp) && dp) {
    size_t dsize = 0;
    struct dirent *newdp = NULL;

    if (selector && (*selector)(dp) == 0) {
        continue;
    }

    if (nfiles == vector_size) {
        struct dirent **newv;
        if (vector_size == 0) {
            vector_size = 10;
        } else {
            vector_size *= 2;
        }

        newv = (struct dirent **) realloc (vector, vector_size * sizeof (struct dirent *));
        if (!newv) {
            return -1;
        }
        vector = newv;
    }

    dsize = sizeof (struct dirent) + ((strlen(dp->d_name) + 1) * sizeof(char));
    newdp = (struct dirent *) malloc(dsize);

    if (newdp == NULL) {
        goto fail;
    }

    vector[nfiles++] = (struct dirent *) memcpy(newdp, dp, dsize);
}

closedir(dirp);
```



On the Accuracy of Spectrum-based Fault Localization*

Rui Abreu   Peter Zoeteweij   Arjan J.C. van Gemund

Software Technology Department
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology
P.O. Box 5031, NL-2600 GA Delft, The Netherlands
{r.f.abreu, p.zoeteweij, a.j.c.vangemund}@tudelft.nl

Abstract

Spectrum-based fault localization shortens the test-diagnose-repair cycle by reducing the debugging effort. As a light-weight automated diagnosis technique it can easily be integrated with existing testing schemes. However, as no model of the system is taken into account, its diagnostic accuracy is inherently limited. Using the Siemens Set benchmark, we investigate this diagnostic accuracy as a function of several parameters (such as quality and quantity of the program spectra collected during the execution of the system), some of which directly relate to test design. Our results indicate that the superior performance of a particular similarity coefficient, used to analyze the program spectra, is largely independent of test design. Furthermore, near-optimal diagnostic accuracy (exonerating about 80% of the blocks of code on average) is already obtained for low-quality error observations and limited numbers of test cases. The influence of the number of test cases is of primary importance for continuous (embedded) processing applications, where only limited observation

those faults that affect the user most can be solved before the release deadline, the efficiency with which faults can be diagnosed and repaired directly influences software reliability. Automated diagnosis can help to improve this efficiency.

Diagnosis techniques are complementary to testing in two ways. First, for tests designed to verify correct behavior, they generate information on the root cause of test failures, focusing the subsequent tests that are required to expose this root cause. Second, for tests designed to expose specific potential root causes, the extra information generated by diagnosis techniques can help to further reduce the set of remaining possible explanations. Given its incremental nature (i.e., taking into account the results of an entire sequence of tests), automated diagnosis alleviates much of the work of selecting tests in the latter category, and can hence have a profound impact on the test-diagnose-repair cycle.

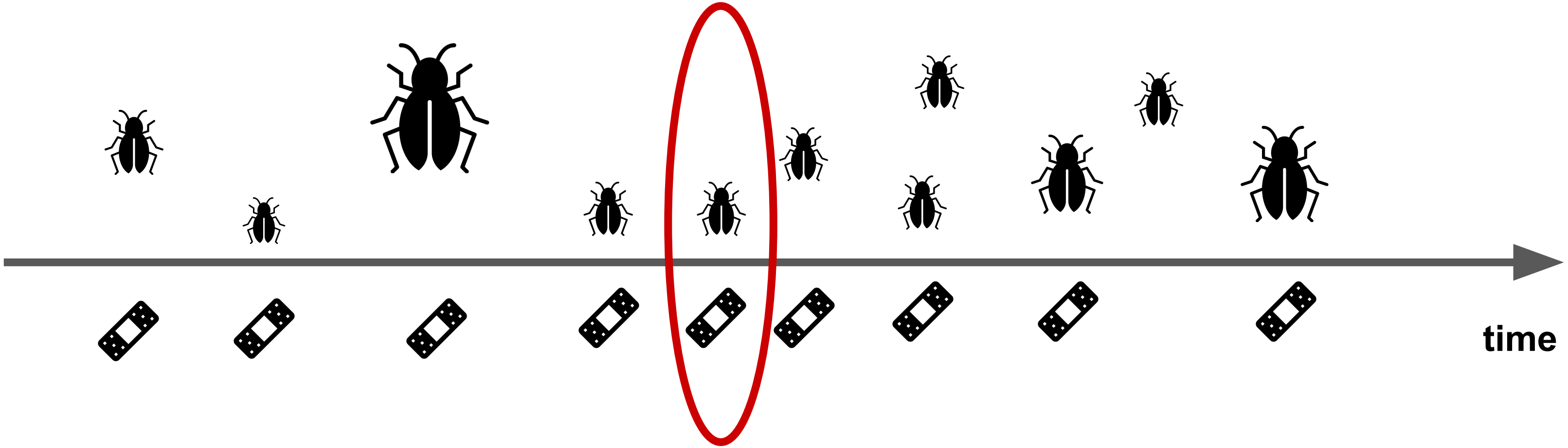
An important part of diagnosis and repair consist in localizing faults, and several tools for automated debugging and systems diagnosis implement an approach

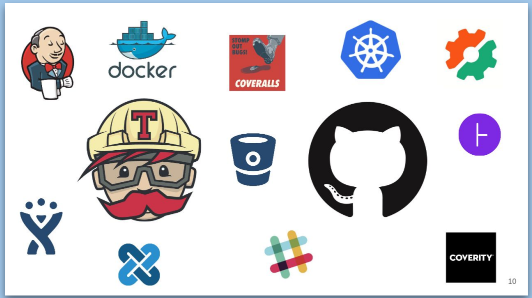
Assumption:
several failing test cases (6 is optimal).

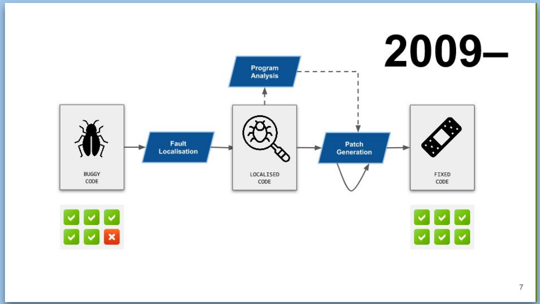
Reality:
usually one failing test.

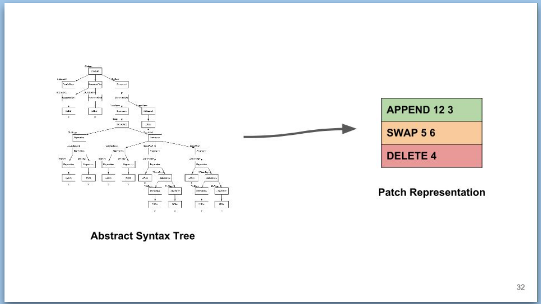
Takes > 12 hours to run GenProg and SearchRepair.

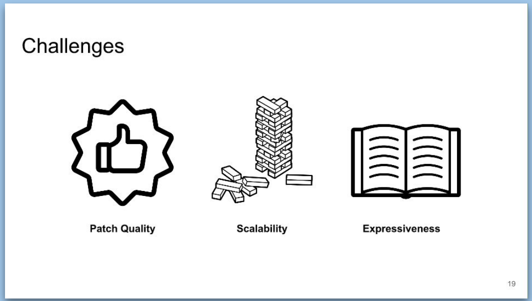
Can we tailor fault localisation to CI-based program repair?

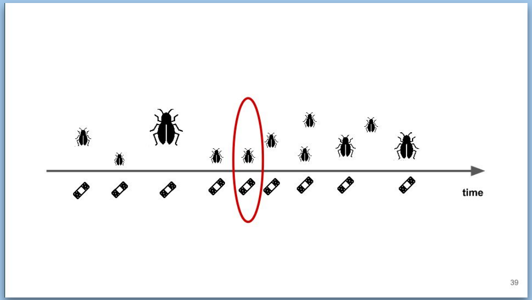

time


docker
COVERALLS
COVERITY
10


2009–
Program Analysis
BUGGY CODE
Fault Localisation
LOCALISED CODE
Patch Generation
FIXED CODE
7


APPEND 12 3
SWAP 5 6
DELETE 4
Patch Representation
Abstract Syntax Tree
32


Challenges
Patch Quality
Scalability
Expressiveness
19


time
39


BugZoo