RUHR-UNIVERSITÄT BOCHUM

Introduction into Automated Program Repair

12.12.2024 – Ringvorlesung – Uni Ulm

Prof. Dr. Yannic Noller
Software Quality group

Software Quality
research group at RUB

About Me

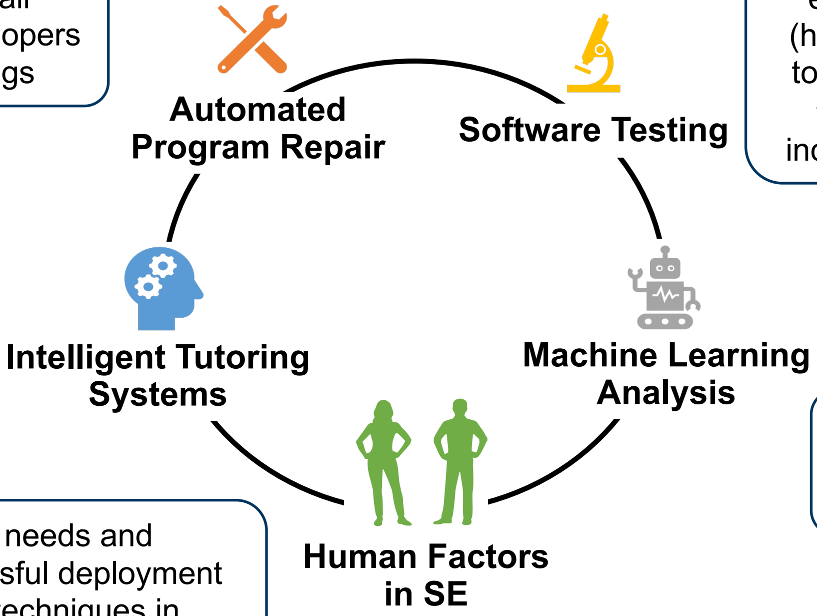
- since **July 2024**: Professor for Computer Science, Ruhr University Bochum
- **Before**:
  - 2023 – 2024: **Singapore** University of Technology and Design (Assistant Professor)
  - 2020 – 2023: National University of **Singapore** (PostDoc, Research Assist. Prof.)
  - 2016 – 2020: PhD student at HU **Berlin**
  - 2010 – 2016: Bachelor and Master in Software Engineering at University of **Stuttgart**
- **Research Interests:**
  - automated software engineering
  - software testing & verification (e.g., symbolic execution and fuzzing)
  - software repair (e.g., semantic-based)

Software Quality Research @ RUB

developing new repair techniques to aid developers in fixing program bugs

exploring and designing (hybrid) testing techniques to systematically generate test inputs that expose incorrect program behavior

how to help CS students learn programming by applying concepts from automated testing and repair to guide the students toward the right solution



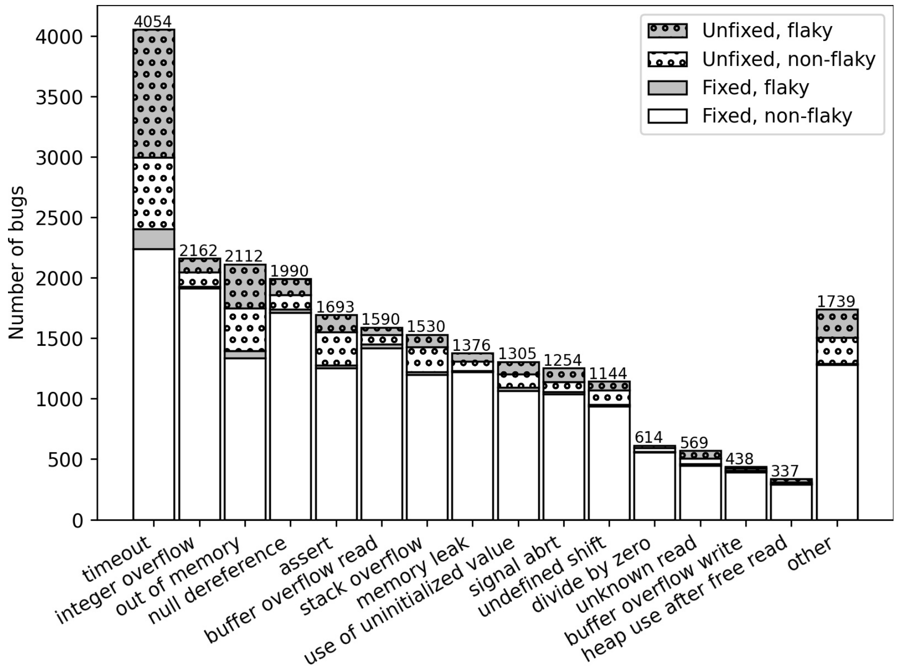
automated analysis, testing, and repairing of machine learning models

studying developer needs and requirements for successful deployment of testing and repair techniques in development practice

https://informatik.rub.de/en/sq/

Bugs are Rising

- A study of over **5000 bugs** found by OSS-Fuzz in the last 5 years
- More than **50%** of the bugs are **security bugs**, e.g., overflows
- Median time to fix non-flaky bugs: approx. 5 days
  - Some remain **unfixed** for long time



Z. Y. Ding and C. Le Goues, "An Empirical Study of OSS-Fuzz Bugs," MSR 2021, https://doi.org/10.1109/MSR52588.2021.00026

RUHR UNIVERSITÄT BOCHUM

Repairs — (often) Simple but not Straightforward

Apache Tomcat

```
@Override
public void run() {
      if (getError() == null) {
       try {
             if (read) {
                 nBytes = getSocket().read(buffers, offset, length);
                 updateLastRead();
             } else {
                 nBytes = getSocket().write(buffers, offset, length);
                 updateLastWrite();
             }
//...
```

Faulty Commit #7040497fa

```
public synchronized void run() {
```

Commit message:

Add sync when processing asynchronous operation in NIO. The NIO poller seems to create some unwanted concurrency, causing rare CI test failures......It doesn't seem right to me that there is concurrency here, **“but it's not hard to add a Sync.”**

Correct Commit #29f060adb

```
@Override
public void run() {
      if (getError() == null) {
      synchronized (this) {
       try {
             if (read) {
                 nBytes = getSocket().read(buffers, offset, length);
                 updateLastRead();
             } else {
                 nBytes = getSocket().write(buffers, offset, length);
                 updateLastWrite();
             }
//...
```

Repairs — (often) Simple but not Straightforward

Buggy Program

How do we fix this?

```
int length, index = 0;
int height[10], breadth[8];
input(length);
while (index < length) {
  height[index] = index + 1;
  ++index;
}
```

Buffer Overflow

length = 11

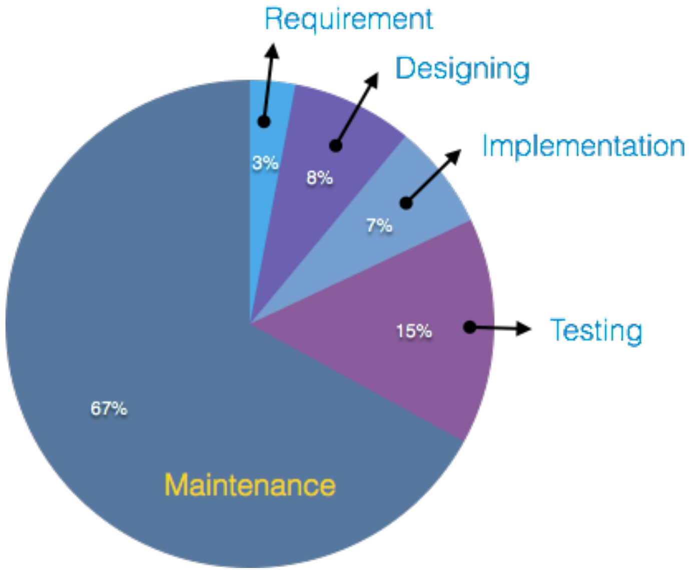
```
while (index >= 0) {
    breadth[index] = index + 1;
    index--;
}
```

Potential Repairs

1. `while (index < length & length < sizeof(height)) {`
2. `while (index < length & index < sizeof(height)) {`
3. `int height[20],…`
4. `if (length > sizeof(height)) { abort();}`

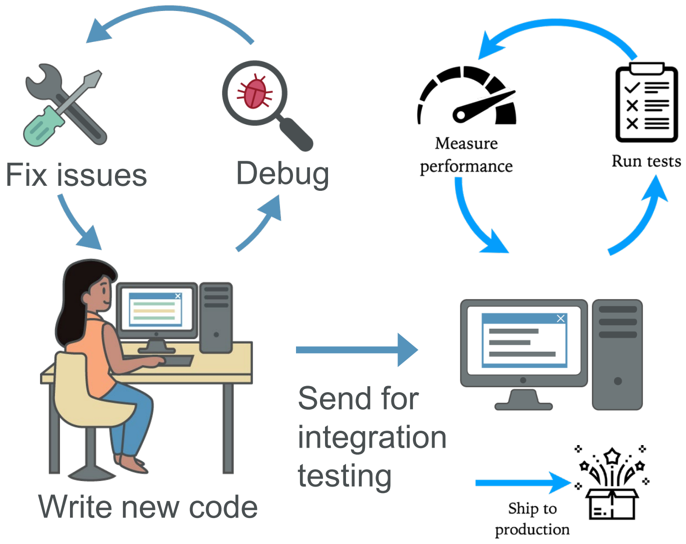
Cost of Repairs

- Maintenance constitutes the major cost of software development
  - It costs ≈ \$312 billion per year



http://www.prweb.com/releases/2013/1/prweb10298185.htm
https://sceweb.sce.uhcl.edu/helm/WEBPAGES-SoftwareEngineering/myfiles/TableContents/Module-13/software_maintenance_overview.html

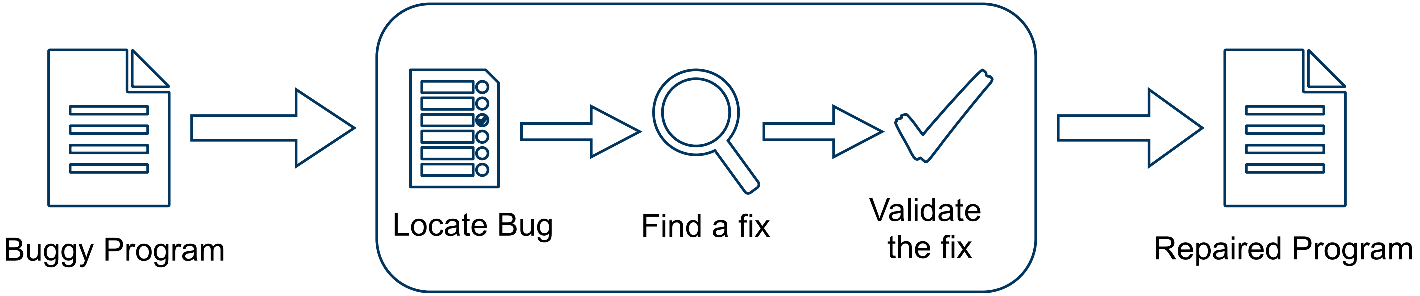
Software Development Life-Cycle



Xiang Gao, Yannic Noller, and Abhik Roychoudhury. "Program repair.“, 2022, https://arxiv.org/abs/2211.12787

RUHR UNIVERSITÄT BOCHUM RUB

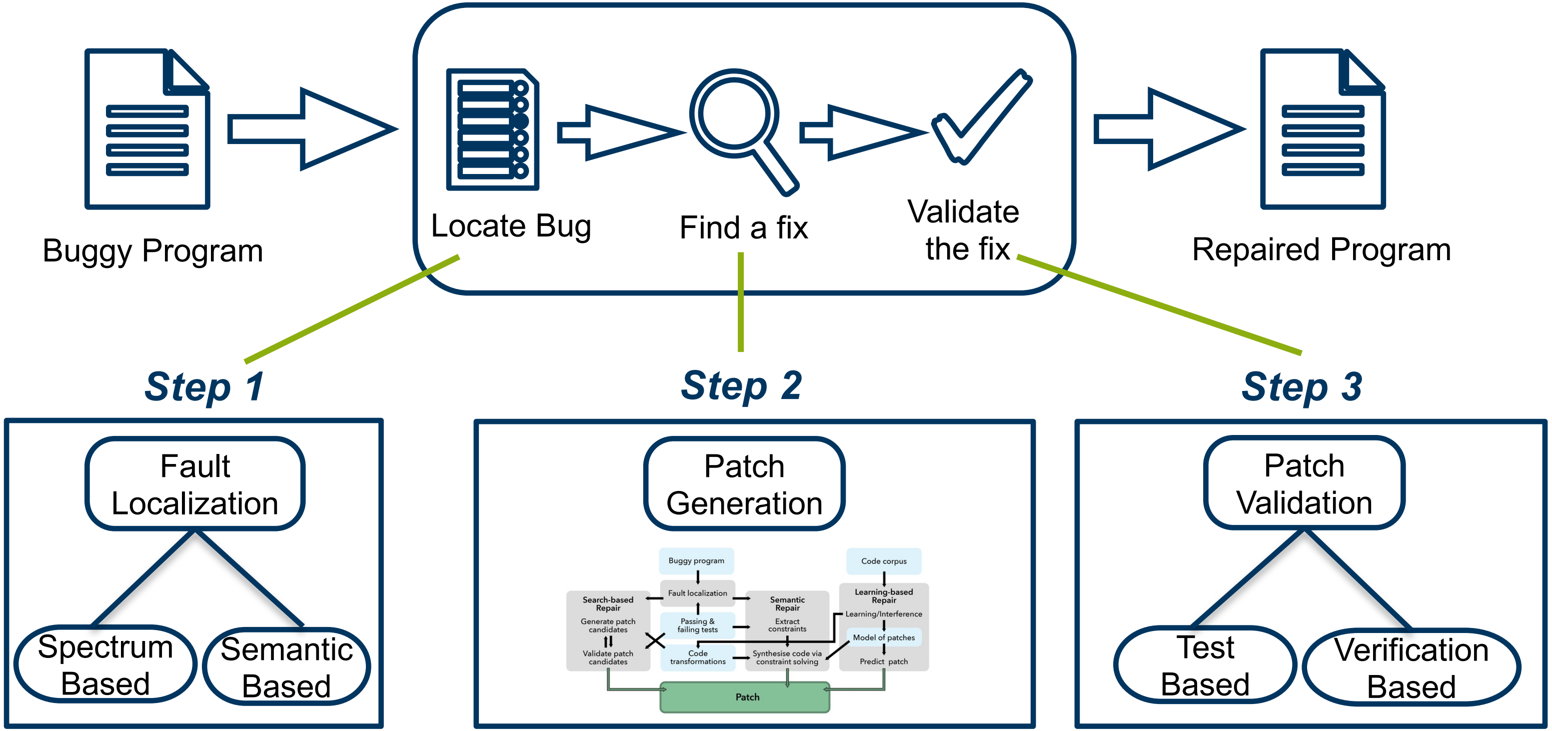
Automated Program Repair (APR)



Roadmap

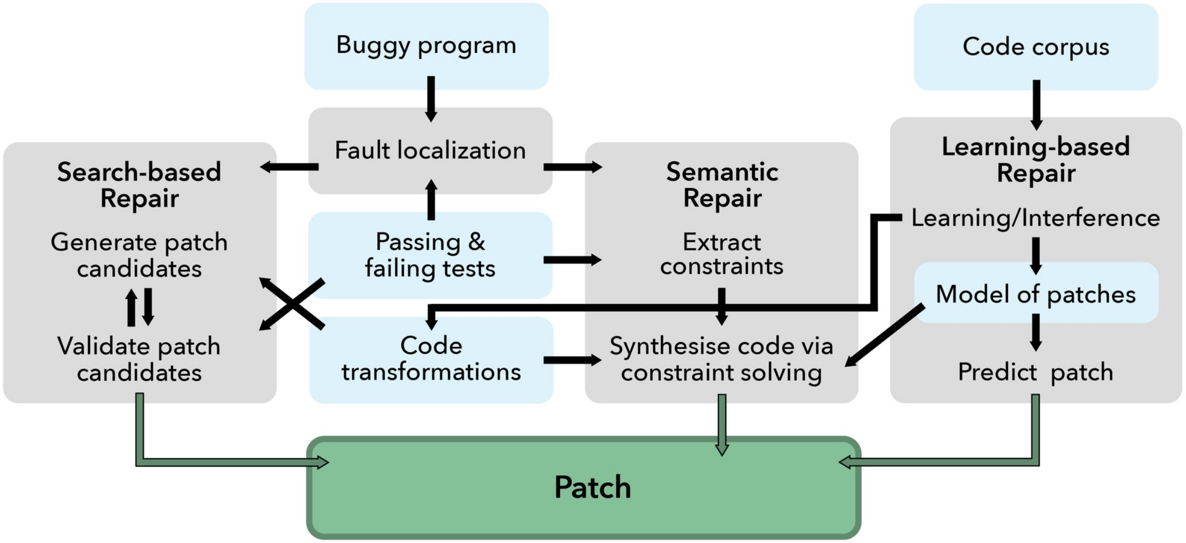
- Brief Introduction
- Fault localization
- Types of Automated Program Repair (APR)
  - Search-based (Generate and Validate)
  - Semantic-based
  - Learning-based
    - APR in the era of Large Language Models (LLM)
- (Repair of Security Vulnerabilities)
- Challenges in Program Repair: Overfitting and Ranking
- Real World applicability of APR tools — Solution and challenges

Main Components



RUHR UNIVERSITÄT BOCHUM RUB

Types of APR



State-of-the-art in Program Repair: Pictorial view derived from Communications of the ACM article 2019.

https://nus-apr.github.io/

Automated Fault Localization

Fault Localization

- **Metric-based**
- Program dependence-based
- Artificial Intelligence-based
- Statistics-based
- Mutation-based

Metric Based Fault Localization

- For each program element, outputs a **suspiciousness** score
- Intuition: Program elements **executed** in **failing** test cases are **likely** to be faulty

- *passed(s)* : number of passing test cases executed the statement s
- *totalpassed*: total number of passed test cases
- *failed(s)*: number of failing test cases executed the statement s
- *totalfailed*: total number of failing test cases

(1/4) Run test cases

(3,3,5)	(4,5,6)	(4,4,4)	(5,3,4)	(2,1,3)	(5,4,9)

```python
def mid(x, y, z):
    m = z
    if (y < z):
        if (x < y):
            m = y
        elif (x < z):
            m = y
    else:
        if (x > y):
            m = y
        elif (x > z):
            m = x
    return m
```

mid(3,3,5) == 3	✔
mid(4,5,6) == 5	✔
mid(4,4,4) == 4	✔
mid(5,3,4) == 4	✔
mid(2,1,3) == 1	✘
mid(5,4,9) == 4	✘

(2/4) Statement Coverage

```python
def mid(x, y, z):
    m = z
    if (y < z):
        if (x < y):
            m = y
        elif (x < z):
            m = y
    else:
        if (x > y):
            m = y
        elif (x > z):
            m = x
    return m
```

mid(3,3,5) = 3

```python
def mid(x, y, z):
    m = z
    if (y < z):
        if (x < y):
            m = y
        elif (x < z):
            m = y
    else:
        if (x > y):
            m = y
        elif (x > z):
            m = x
    return m
```

mid(5,3,4) = 4

```python
def mid(x, y, z):
    m = z
    if (y < z):
        if (x < y):
            m = y
        elif (x < z):
            m = y
    else:
        if (x > y):
            m = y
        elif (x > z):
            m = x
    return m
```

mid(2,1,3) = 1

(2/4) Compute Statement Coverage

Line	Statement	(3,3,5)	(4,5,6)	(4,4,4)	(5,3,4)	(2,1,3)	(5,4,9)	Passed(s)	Failed(s)
2	m=z	●	●	●	●	●	●	4	2
3	if (y < z)	●	●	●	●	●	●	4	2
4	if (x < y)	●	●		●	●	●	3	2
5	m=y		●					1	0
6	elif (x < z)	●			●	●	●	2	2
7	m=y	●				●	●	1	2
8	Else							0	0
9	if (x > y)			●				1	0
10	m=y							0	0
11	elif(x > z)			●				1	0
12	m=x							0	0
13	return m	●	●	●	●	●	●	4	2
		PASS	PASS	PASS	PASS	FAIL	FAIL		

RUHR UNIVERSITÄT BOCHUM RUB

(3/4) Compute Suspiciousness score

- Different metrics to compute suspiciousness score
  - Tarantula
  - Occhia
  - Op2
  - Barinel
  - Star
  - ...

(3/4) Tarantula

$$S(s) = \frac{failed(s)/totalfailed}{failed(s)/totalfailed + passed(s)/totalpassed}$$

- First proposed technique for the fault localization

Line	Statement	(3,3,5)	(4,5,6)	(4,4,4)	(5,3,4)	(2,1,3)	(5,4,9)	Score
2	m=z	●	●	●	●	●	●	0.50
3	If (y < z)	●	●	●	●	●	●	0.50
4	If (x < y)	●	●		●	●	●	0.57
5	m=y		●					0.00
6	Elif (x < z)	●			●	●	●	0.67
7	m=y	●				●	●	0.80
8	Else							
9	If (x > y)			●				0.00
10	m=y							
11	elif(x > z)			●				0.00
12	m=x							
13	Return m	●	●	●	●	●	●	0.50
		PASS	**PASS**	**PASS**	**PASS**	**FAIL**	**FAIL**	

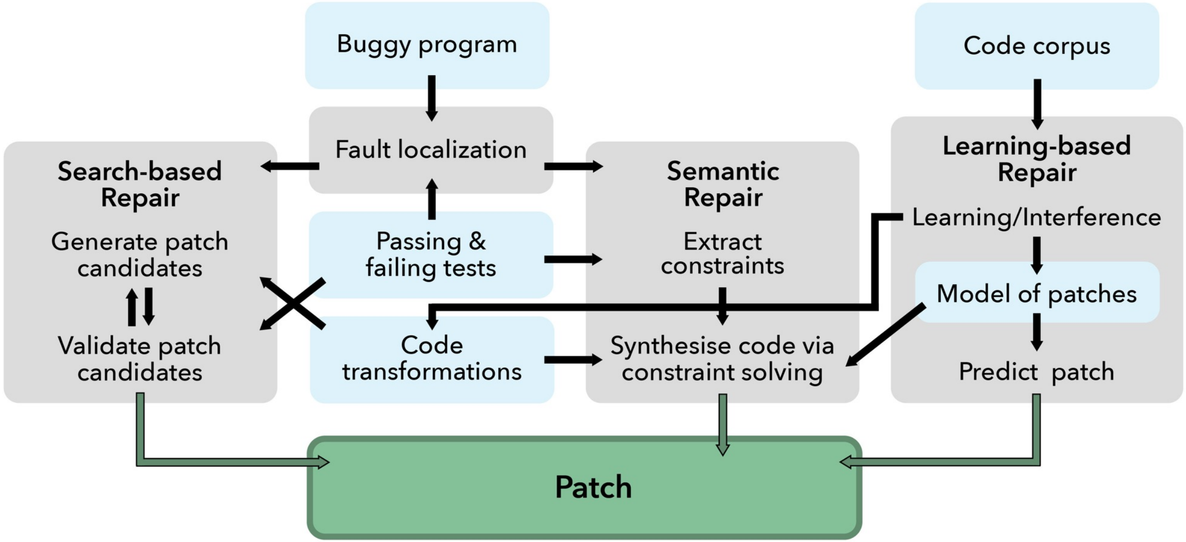
(4/4) Prioritising Statements

- A Program Repair technique requires to know which statement it has to fix first
- Solution: Prioritise by the suspicion score

```
def mid(x, y, z):
    m = z                    0.5
    if (y < z):              0.5
        if (x < y):          0.57
            m = y
        elif (x < z):        0.67
            m = y            0.8
    else:
        if (x > y):
            m = y
        elif (x > z):
            m = x
    return m                 0.5
```

Patch Generation

- **With a list of suspicious locations, the next step is to correct them!**
- Multiple approaches exist:



Search-based APR

GenProg: A Generic Method for Automatic Software Repair

Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, *Senior Member*, *IEEE*, and Westley Weimer

Abstract—This paper describes GenProg, an automated method for repairing defects in off-the-shelf, legacy programs without formal specifications, program annotations, or special coding practices. GenProg uses an extended form of genetic programming to evolve a program variant that retains required functionality but is not susceptible to a given defect, using existing test suites to encode both the defect and required functionality. Structural differencing algorithms and delta debugging reduce the difference between this variant and the original program to a minimal repair. We describe the algorithm and report experimental results of its success on 16 programs totaling 1.25 M lines of C code and 120K lines of module code, spanning eight classes of defects, in 357 seconds, on average. We analyze the generated repairs qualitatively and quantitatively to demonstrate that the process efficiently produces evolved programs that repair the defect, are not fragile input memorizations, and do not lead to serious degradation in functionality.

Index Terms—Automatic programming, corrections, testing and debugging.

1 Introduction

Software quality is a pernicious problem. Mature software projects are forced to ship with both known and unknown bugs [1] because the number of outstanding software defects typically exceeds the resources available to address them [2]. Software maintenance, of which bug repair is a major component [3], [4], is time-consuming and expensive, accounting for as much as 90 percent of the cost of a software project [5] at a total cost of up to $70 billion per year in the US [6], [7]. Put simply: Bugs are ubiquitous, and finding and repairing them are difficult, time-consuming, and manual processes.

Techniques for automatically detecting software flaws include intrusion detection [8], model checking and lightweight static analyses [9], [10], and software diversity methods [11], [12]. However, detecting a defect is only half of the story: Once identified, a bug must still be repaired. As the scale of software deployments and the frequency of defect reports increase [13], some portion of the repair problem must be addressed automatically.

This paper describes and evaluates Genetic Program Repair ("GenProg"), a technique that uses existing test cases to automatically generate repairs for real-world bugs in off-the-shelf, legacy applications. We follow Rinard et al. [14] in defining a *repair* as a patch consisting of one or more code changes that, when applied to a program, cause it to pass a set of test cases (typically including both tests of required behavior as well as a test case encoding the bug). The test cases may be human written, taken from a regression test suite, steps to reproduce an error, or generated automatically. We use the terms "repair" and "patch" interchangeably. GenProg does not require formal specifications, program annotations, or special coding practices. GenProg's approach is generic, and the paper reports results demonstrating that GenProg can successfully repair several types of defects. This contrasts with related approaches which repair only a specific type of defect (such as buffer overruns [15], [16]).

GenProg takes as input a program with a defect and a set of test cases. GenProg may be applied either to the full program source or to individual modules. It uses *genetic programming* (GP) to search for a program variant that retains required functionality but is not vulnerable to the defect in question. GP is a stochastic search method inspired by biological evolution that discovers computer programs tailored to a particular task [17], [18]. GP uses computational analogs of biological mutation and crossover to generate new program variations, which we call *variants*. A user-defined *fitness function* evaluates each variant; GenProg uses the input test cases to evaluate the fitness, and individuals with high fitness are selected for continued evolution. This GP process is successful when it produces a variant that passes all tests encoding the required behavior and does not fail those encoding the bug. Although GP has solved an impressive range of problems (e.g., [19]), it has not previously been used either to evolve off-the-shelf legacy software or to patch real-world vulnerabilities, despite various proposals directed at automated error repair, e.g., [20].

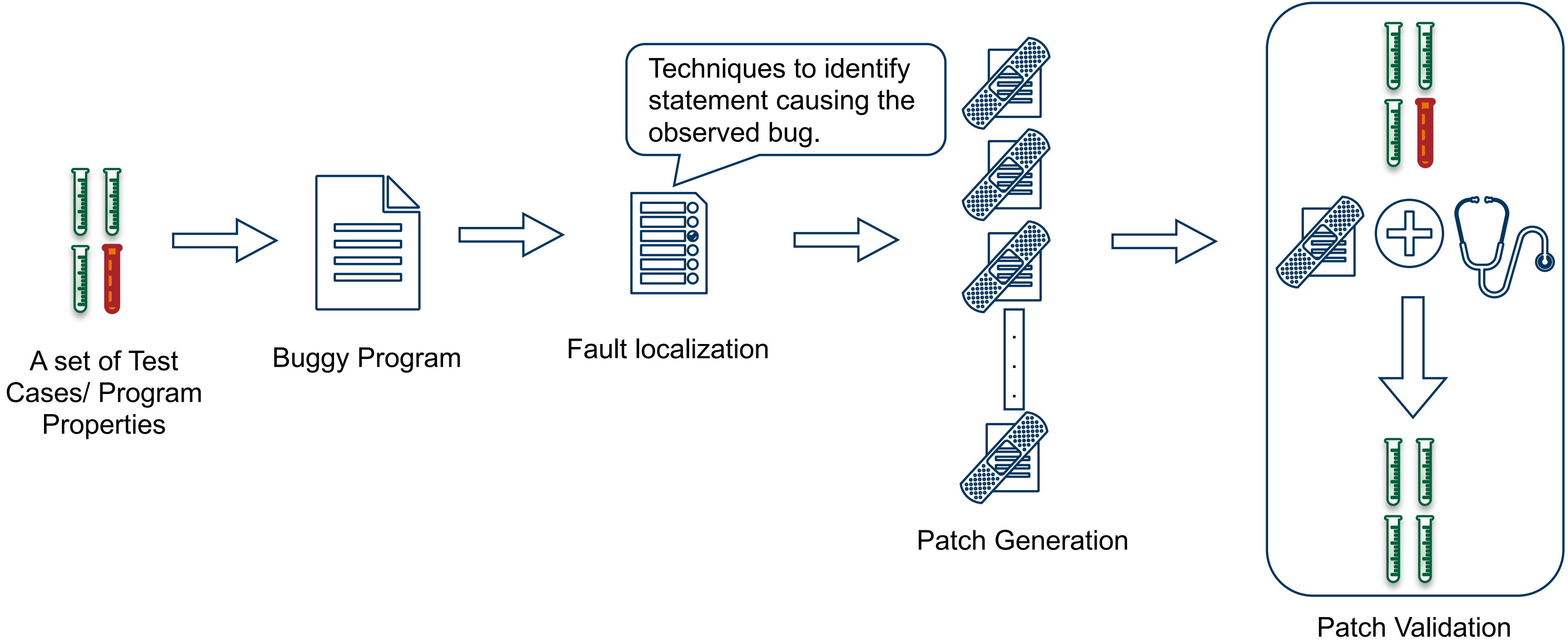
A significant impediment for GP efforts to date has been the potentially infinite space that must be searched to find a correct program. We introduce three key innovations to address this longstanding problem [21]. First, GenProg operates at the *statement level* of a program's abstract syntax tree (AST), increasing the search granularity. Second, we hypothesize that a program that contains an error in one area likely implements the correct behavior elsewhere [22]. Therefore, GenProg uses only statements from the program

- *C. Le Goues and W. Weimer are with the Department of Computer Science, University of Virginia, 85 Engineer's Way, PO Box 400740, Charlottesville, VA 22904-4740. E-mail: {legoues, weimer}@cs.virginia.edu.*
- *T. Nguyen and S. Forrest are with the Department of Computer Science, University of New Mexico, MSC01 1130, 1 University of New Mexico, Albuquerque, NM 87131-0001. E-mail: {tnguyen, forrest}@cs.unm.edu.*

Manuscript received 16 Mar. 2010; revised 6 Oct. 2010; accepted 21 Sept. 2011; published online 30 Sept. 2011.
Recommended for acceptance by J.M. Atlee and P. Inverardi.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSESI-2010-03-0078.
Digital Object Identifier no. 10.1109/TSE.2011.104.

0098-5589/12/$31.00 © 2012 IEEE Published by the IEEE Computer Society

Search Based (Generate & Validate) APR



RUHR UNIVERSITÄT BOCHUM RUB

Search-Based APR Tools

- **GenProg: A generic method for APR**
- SPR: Staged Program Repair with Condition Synthesis
- History Driven Program repair
- Prophet: Automatic patch generation by leaning correct code

- … and many more

GenProg

- Based on **Genetic Programming**
  - A programming model for **evolving** programs
  - Ideology and terminology of **biological evolution** to address program evolution
  - Starting from a population of **unfit** (buggy) program — apply operations analogous to **natural genetic processes** — define a fitness function to evaluate evolved program
  - **Fitness function** evaluates the quality of an evolved program
- Given an input test suite of passing and failing test, creates mutated programs (repairs) that solves the failing test

GenProg: A Generic Method for Automatic Software Repair

Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, *Senior Member, IEEE*, and Westley Weimer

Abstract—This paper describes GenProg, an automated method for repairing defects in off-the-shelf, legacy programs without formal specifications, program annotations, or special coding practices. GenProg uses an extended form of genetic programming to evolve a program variant that retains required functionality but is not susceptible to a given defect, using existing test suites to encode both the defect and required functionality. Structural differencing algorithms and delta debugging reduce the difference between this variant and the original program to a minimal repair. We describe the algorithm and report experimental results of its success on 16 programs totaling 1.25 M lines of C code and 120K lines of module code, spanning eight classes of defects, in 357 seconds, on average. We analyze the generated repairs qualitatively and quantitatively to demonstrate that the process efficiently produces evolved programs that repair the defect, are not fragile input memorizations, and do not lead to serious degradation in functionality.

Index Terms—Automatic programming, corrections, testing and debugging.

1 Introduction

Software quality is a pernicious problem. Mature software projects are forced to ship with both known and unknown bugs [1] because the number of outstanding software defects typically exceeds the resources available to address them [2]. Software maintenance, of which bug repair is a major component [3], [4], is time-consuming and expensive, accounting for as much as 90 percent of the cost of a software project [5] at a total cost of up to $70 billion per year in the US [6], [7]. Put simply: Bugs are ubiquitous, and finding and repairing them are difficult, time-consuming, and manual processes.

Techniques for automatically detecting software flaws include intrusion detection [8], model checking and lightweight static analyses [9], [10], and software diversity methods [11], [12]. However, detecting a defect is only half of the story: Once identified, a bug must still be repaired. As the scale of software deployments and the frequency of defect reports increase [13], some portion of the repair problem must be addressed automatically.

This paper describes and evaluates Genetic Program Repair ("GenProg"), a technique that uses existing test cases to automatically generate repairs for real-world bugs in off-the-shelf, legacy applications. We follow Rinard et al. [14] in defining a *repair* as a patch consisting of one or more code changes that, when applied to a program, cause it to pass a set of test cases (typically including both tests of required behavior as well as a test case encoding the bug). The test cases may be human written, taken from a regression test suite, steps to reproduce an error, or generated automatically. We use the terms "repair" and "patch" interchangeably. GenProg does not require formal specifications, program annotations, or special coding practices. GenProg's approach is generic, and the paper reports results demonstrating that GenProg can successfully repair several types of defects. This contrasts with related approaches which repair only a specific type of defect (such as buffer overruns [15], [16]).

GenProg takes as input a program with a defect and a set of test cases. GenProg may be applied either to the full program source or to individual modules. It uses *genetic programming* (GP) to search for a program variant that retains required functionality but is not vulnerable to the defect in question. GP is a stochastic search method inspired by biological evolution that discovers computer programs tailored to a particular task [17], [18]. GP uses computational analogs of biological mutation and crossover to generate new program variations, which we call *variants*. A user-defined *fitness function* evaluates each variant; GenProg uses the input test cases to evaluate the fitness, and individuals with high fitness are selected for continued evolution. This GP process is successful when it produces a variant that passes all tests encoding the required behavior and does not fail those encoding the bug. Although GP has solved an impressive range of problems (e.g., [19]), it has not previously been used either to evolve off-the-shelf legacy software or to patch real-world vulnerabilities, despite various proposals directed at automated error repair, e.g., [20].

A significant impediment for GP efforts to date has been the potentially infinite space that must be searched to find a correct program. We introduce three key innovations to address this longstanding problem [21]. First, GenProg operates at the *statement level* of a program's abstract syntax tree (AST), increasing the search granularity. Second, we hypothesize that a program that contains an error in one area likely implements the correct behavior elsewhere [22]. Therefore, GenProg uses only statements from the program

- *C. Le Goues and W. Weimer are with the Department of Computer Science, University of Virginia, 85 Engineer's Way, PO Box 400740, Charlottesville, VA 22904-4740. E-mail: {legoues, weimer}@cs.virginia.edu.*
- *T. Nguyen and S. Forrest are with the Department of Computer Science, University of New Mexico, MSC01 1130, 1 University of New Mexico, Albuquerque, NM 87131-0001. E-mail: {tnguyen, forrest}@cs.unm.edu.*

Manuscript received 16 Mar. 2010; revised 6 Oct. 2010; accepted 21 Sept. 2011; published online 30 Sept. 2011.
Recommended for acceptance by J.M. Atlee and P. Inverardi.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSESI-2010-03-0078.
Digital Object Identifier no. 10.1109/TSE.2011.104.

0098-5589/12/$31.00 © 2012 IEEE Published by the IEEE Computer Society

https://doi.org/10.1109/TSE.2011.104

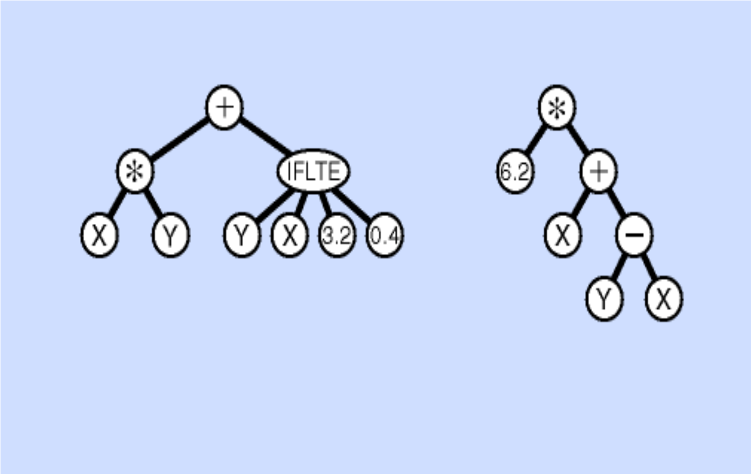
RUHR UNIVERSITÄT BOCHUM RUB

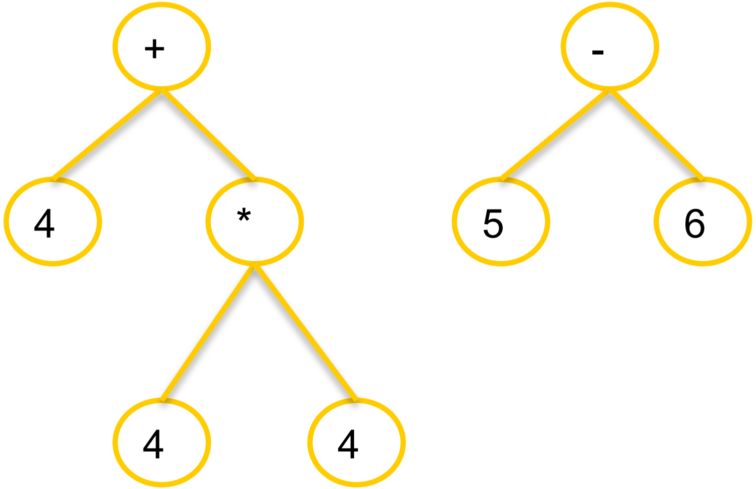
Initial Population: Selection

- Selection of individual to serve as **parents** for **next generation**
- Aim to select **better performing** individuals
- Various selection techniques
  - **Stochastic universal sampling**— probability of selection of a parent is **directly proportional** to its fitness
  - **Tournament selection**—a small subset of population are randomly selected (by a tournament) and the **most fit** member of this **subset** is selected for next generation

Variants Generation: Crossover

- Program represented as tree structure (mostly as AST)
- **Swap** random parts in parents to produce **new** children

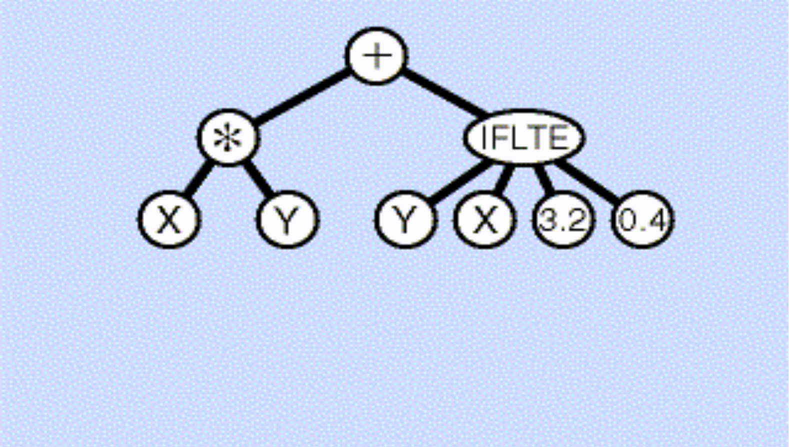


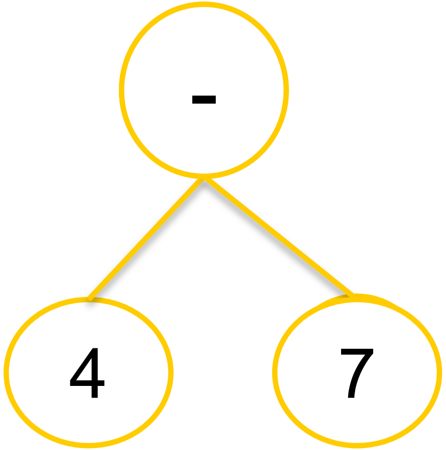


By W102102 - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=106389498

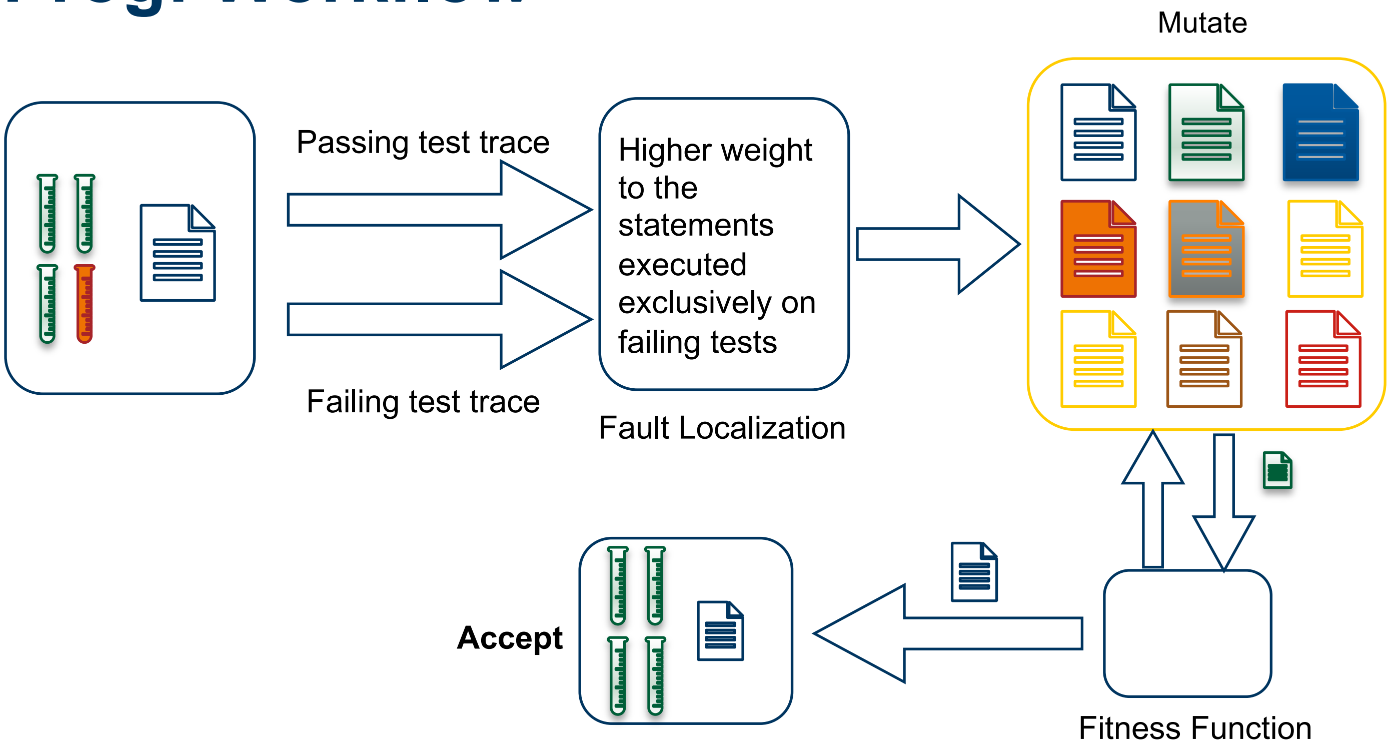
Variants Generation: Mutation

- Various types of mutations (**syntactically correct**)
- Intuitively, update (**insert, remove, or delete**) a parent node to obtain a new child





GenProg: Workflow



RUHR UNIVERSITÄT BOCHUM RUB

Fault Localization

- Any statement **executed** by a **negative** test case contains an **initial** weight of 1.0
- **Other** statements are assigned weight 0.0
  - these are never modified, i.e., these are consider not faulty
- The **initial weight** of statements executed by a negative test case is **modified** if they are also executed by a positive test case
- Goal is to **penalize** statements that are more unique to negative tests
- **No** additional weights for statements frequencies (e.g., in a loop)

Mutation

Input: Program P to be mutated.
Input: Path $Path_P$ of interest.
Output: Mutated program variant.

$\textbf{for all}\ \langle stmt_i, prob_i \rangle \in Path_P\ \textbf{do}$
$\quad \textbf{if}\ \mathsf{rand}(0,1) \leq prob_i \land \mathsf{rand}(0,1) \leq W_{mut}\ \textbf{then}$
$\quad\quad \textbf{let}\ op = \mathsf{choose}(\{\mathsf{insert}, \mathsf{swap}, \mathsf{delete}\})$
$\quad\quad \textbf{if}\ op = \mathsf{swap}\ \textbf{then}$
$\quad\quad\quad \textbf{let}\ stmt_j = \mathsf{choose}(P)$
$\quad\quad\quad Path_P[i] \leftarrow \langle stmt_j, prob_i \rangle$
$\quad\quad \textbf{else if}\ op = \mathsf{insert}\ \textbf{then}$
$\quad\quad\quad \textbf{let}\ stmt_j = \mathsf{choose}(P)$
$\quad\quad\quad Path_P[i] \leftarrow \langle \{stmt_i; stmt_j\}, prob_i \rangle$
$\quad\quad \textbf{else if}\ op = \mathsf{delete}\ \textbf{then}$
$\quad\quad\quad Path_P[i] \leftarrow \langle \{\}, prob_i \rangle$
$\quad\quad \textbf{end if}$
$\quad \textbf{end if}$
$\textbf{end for}$
$\textbf{return}\ \langle P, Path_P \rangle$

Ranked Fault locations

Mutation Operators

Choose a statement from the same program

Fitness Function

- Evaluate the **quality** of a program variant
- Each **successful** positive test is weighted by W_{PosT}
- Each **successful** negative test is weighted by W_{NegT}
- Program variants that do **not** compile have **zero** fitness
- GenProg encode W_{PosT} as **1** and W_{NegT} as **10** in their evaluation setup

$$\begin{aligned} \text{fitness}(P) = {} & W_{PosT} \times |\{t \in PosT \mid P \text{ passes } t\}| \\ & + W_{NegT} \times |\{t \in NegT \mid P \text{ passes } t\}|. \end{aligned}$$

RUHR UNIVERSITÄT BOCHUM RUB

Crossover

Input: Parent programs P and Q.
Input: Paths $Path_P$ and $Path_Q$.
Output: Two new child program variants C and D.

$cutoff \leftarrow \mathsf{choose}(|Path_P|)$
$C, Path_C \leftarrow \mathsf{copy}(P, Path_P)$
$D, Path_D \leftarrow \mathsf{copy}(Q, Path_Q)$
for $i = 1$ to $|Path_P|$ **do**
  if $i > cutoff$ **then**
    $Path_C[i] \leftarrow Path_Q[i]$
    $Path_D[i] \leftarrow Path_P[i]$
  end if
end for
return $\langle C, Path_C \rangle, \langle D, Path_D \rangle$

swap after the cutoff point

C ← Q
D ← P

Crossover

GenProg: High level Pseudocode

Input: Program P to be repaired.
Input: Set of positive test cases $PosT$.
Input: Set of negative test cases $NegT$.
Input: Fitness function f.
Input: Variant population size pop_size.
Output: Repaired program variant.

$Path_{PosT} \leftarrow \bigcup_{p \in PosT}$ statements visited by $P(p)$
$Path_{NegT} \leftarrow \bigcup_{n \in NegT}$ statements visited by $P(n)$
$Path \leftarrow$ set_weights$(Path_{NegT}, Path_{PosT})$
$Popul \leftarrow$ initial_population$(P,$ pop_size$)$
repeat
  $Viable \leftarrow \{\langle P, Path_P \rangle \in Popul \mid f(P) > 0\}$
  $Popul \leftarrow \emptyset$
  $NewPop \leftarrow \emptyset$
  for all $\langle p_1, p_2 \rangle \in$ select$(Viable, f,$ pop_size$/2)$ **do**
    $\langle c_1, c_2 \rangle \leftarrow$ crossover(p_1, p_2)
    $NewPop \leftarrow NewPop \cup \{p_1, p_2, c_1, c_2\}$
  end for
  for all $\langle V, Path_V \rangle \in NewPop$ **do**
    $Popul \leftarrow Popul \cup \{$mutate$(V, Path_V)\}$
  end for
until $f(V) =$ max_fitness for some V contained in $Popul$
return minimize$(V, P, PosT, NegT)$

Localize and assign weight

pop_size = 40

Create new population using crossover

Mutate the new population

Example

```
receive(packet);
switch (packet.value){
    case 'DHCP':
        data = packet.value;
        break;
    case 'IMAP':
        data = packet.value;
        break;
  default:
        data = packet.value;

        break;
}
...
send(packet, flag);
...
```

Delete free(packet);

Limitations

- **Overfitting** of test cases — repairs that **only** pass a particular test suite
- Generated repairs may **delete** the functionality — pass the test case by removing the functionality
- **Limited** search space

Template-based Repair

- Pre-defined repair patterns
- **Replace** a suspicious program location with defined repair pattern

Insert Null point checker

```
FP2.1: + if (exp != null) {
           ...exp...; ......
       + }
FP2.2: + if (exp == null) return DEFAULT_VALUE;
             ...exp...;
FP2.3: + if (exp == null) exp = exp1;
              ...exp...;
FP2.4: + if (exp == null) continue;
              ...exp...;
FP2.5: + if (exp == null)
       +   throw new IllegalArgumentException(...);
         ...exp...;
```

Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: revisiting template-based automated program repair. https://doi.org/10.1145/3293882.3330577

Semantic-based APR

SemFix: Program Repair via Semantic Analysis

Hoang Duong Thien Nguyen Dawei Qi Abhik Roychoudhury
School of Computing, National University of Singapore, Singapore
{hoangdtn,dawei,abhik}@comp.nus.edu.sg

Satish Chandra
IBM Research, USA
satishchandra@us.ibm.com

***Abstract*—Debugging consumes significant time and effort in any major software development project. Moreover, even after the root cause of a bug is identified, fixing the bug is non-trivial. Given this situation, automated program repair methods are of value. In this paper, we present an automated repair method based on symbolic execution, constraint solving and program synthesis. In our approach, the requirement on the repaired code to pass a given set of tests is formulated as a constraint. Such a constraint is then solved by iterating over a layered space of repair expressions, layered by the complexity of the repair code. We compare our method with recently proposed genetic programming based repair on SIR programs with seeded bugs, as well as fragments of GNU Coreutils with real bugs. On these subjects, our approach reports a higher success-rate than genetic programming based repair, and produces a repair faster.**

I. Introduction

Bug fixing continues to be a mostly manual, time consuming, and therefore expensive activity in software development. Therefore, automated techniques to repair buggy programs can be of tremendous value. In particular, given that compute cycles are cheap and abundant, it makes sense to investigate techniques that help shift the "heavy lifting" of program repair from the human to the computer. While a programmer might not blindly trust a computer-generated fix to her code, her task can become considerably easier: rather than figure out a fix, just verify that an automatically generated fix is correct. Not surprisingly, researchers have recently started looking into automated program repair tools [1]–[3].

We focus on general purpose programs, for which a test suite is available as a way to tell whether the program is working correctly (i.e. it passes all the tests) or not (i.e. there exists a failing test), but otherwise no formal specification of correct behavior is available; this is generally the case in practice (by contrast, kernels that manipulate data structures often do have specifications, and automatic repair on data structure programs have been well studied, for example see [4], [5]). A successful repair would be a modification of the program such that it passes all the tests in the test suite.

One of the most successful techniques in recent work that works on general programs is based on syntactic search. The premise behind this technique is that, once we know where the defective expression is in the program, a correct expression may be present syntactically at another place in the program, so it is a matter of searching over a space of replacements from among existing expressions.[1] The technique uses genetic programming technique for searching over this space, and has been shown to work for large programs [6]. The limitation of this technique is that the correct expression should be present in the program; the technique cannot "synthesize" an appropriate expression from variables and constants.

An obvious response to the limitation would be a search over a space of syntactic expressions, without consideration of whether those expressions appear elsewhere in the program. Such an approach would be more in the flavor of *sketching* [7], [8]. However, unless the space of repair expression is fixed upfront (possibly as a set of templates), such a technique will not work. Furthermore, as our experiments show, enumerating over the set of possible repair templates is inefficient.

In this paper, we explore a constraint based semantic approach towards program repair. The repair constraints are generated by our desire to have the repaired program pass the given test cases. Thus, given a program location to be fixed, we derive constraints on the expression to appear in the program location, in order to have the changed program pass all the given tests. The repair constraints are generated via (controlled) symbolic execution and the expression to be repaired is obtained via program synthesis. We report that, for certain kinds of program and bugs, the semantics-based approach can not only have a higher-success rate than a syntactic search-based approach, but also be able to produce a repair faster. At the same time, we do believe that symbolic execution imposes certain scalability limitations on the size of programs we can handle.

Our approach is a combination of three existing techniques.

- *Fault isolation*, i.e. where to fix the problem. The technique uses the ranking produced by a statistical fault isolation [9] tool (it shares this step with the search-based techniques.) Our approach examines *one buggy statement at a time* from a ranked suspicion report of statements.
- *Statement-level specification inference*. We automatically discover the correct specification of the buggy statement. We use an idea similar to the one used in angelic debugging [10] in converting an expression to a non-deterministic expression. This step allows us to create, for each input to the buggy statement, the output that would have resulted in the test passing.
- *Program synthesis*. The third idea is to use component-based synthesis idea [11] to synthesize an expression that conforms to the specification discovered before.

The inter-play of the second and third steps is the primary novelty of our repair tool. The statement-level specification narrows the search space significantly, and sets up the problem

[1] This is an oversimplification, but broadly speaking this is the idea.

978-1-4673-3076-3/13/$31.00 © 2013 IEEE 772 ICSE 2013, San Francisco, CA, USA

Concolic Program Repair

Ridwan Shariffdeen*
National University of Singapore
Singapore
ridwan@comp.nus.edu.sg

Yannic Noller*
National University of Singapore
Singapore
yannic.noller@acm.org

Lars Grunske
Humboldt-Universität zu Berlin
Germany
grunske@informatik.hu-berlin.de

Abhik Roychoudhury
National University of Singapore
Singapore
abhik@comp.nus.edu.sg

Abstract

Automated program repair reduces the manual effort in fixing program errors. However, existing repair techniques modify a buggy program such that it passes given tests. Such repair techniques do not discriminate between correct patches and patches that overfit the available tests (breaking untested but desired functionality). We propose an integrated approach for detecting and discarding overfitting patches via systematic co-exploration of the patch space and input space. We leverage concolic path exploration to systematically traverse the input space (and generate inputs), while ruling out significant parts of the patch space. Given a long enough time budget, this approach allows a significant reduction in the pool of patch candidates, as shown by our experiments. We implemented our technique in the form of a tool called 'CPR' and evaluated its efficacy in reducing the patch space by discarding overfitting patches from a pool of plausible patches. We evaluated our approach for fixing real-world software vulnerabilities and defects, for fixing functionality errors in programs drawn from SV-COMP benchmarks used in software verification, as well as for test-suite guided repair. In our experiments, we observed a patch space reduction due to our concolic exploration of up to 74% for fixing software vulnerabilities and up to 63% for SV-COMP programs. Our technique presents the viewpoint of *gradual correctness* — repair run over longer time leads to less overfitting fixes.

***CCS Concepts:* • Software and its engineering → Software testing and debugging.**

*Joint first authors

PLDI '21, June 20–25, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8391-2/21/06...$15.00
https://doi.org/10.1145/3453483.3454051

Keywords: program repair, symbolic execution, program synthesis, patch overfitting

ACM Reference Format:
Ridwan Shariffdeen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. 2021. Concolic Program Repair. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3453483.3454051

1 Introduction

Automated Program Repair [14, 24] is an emerging technology which seeks to rectify errors or vulnerabilities in software automatically. There are various applications of automated repair, including improving programmer productivity, reducing exposure to software security vulnerabilities, producing self-healing software systems, and even enabling intelligent tutoring systems for teaching programming.

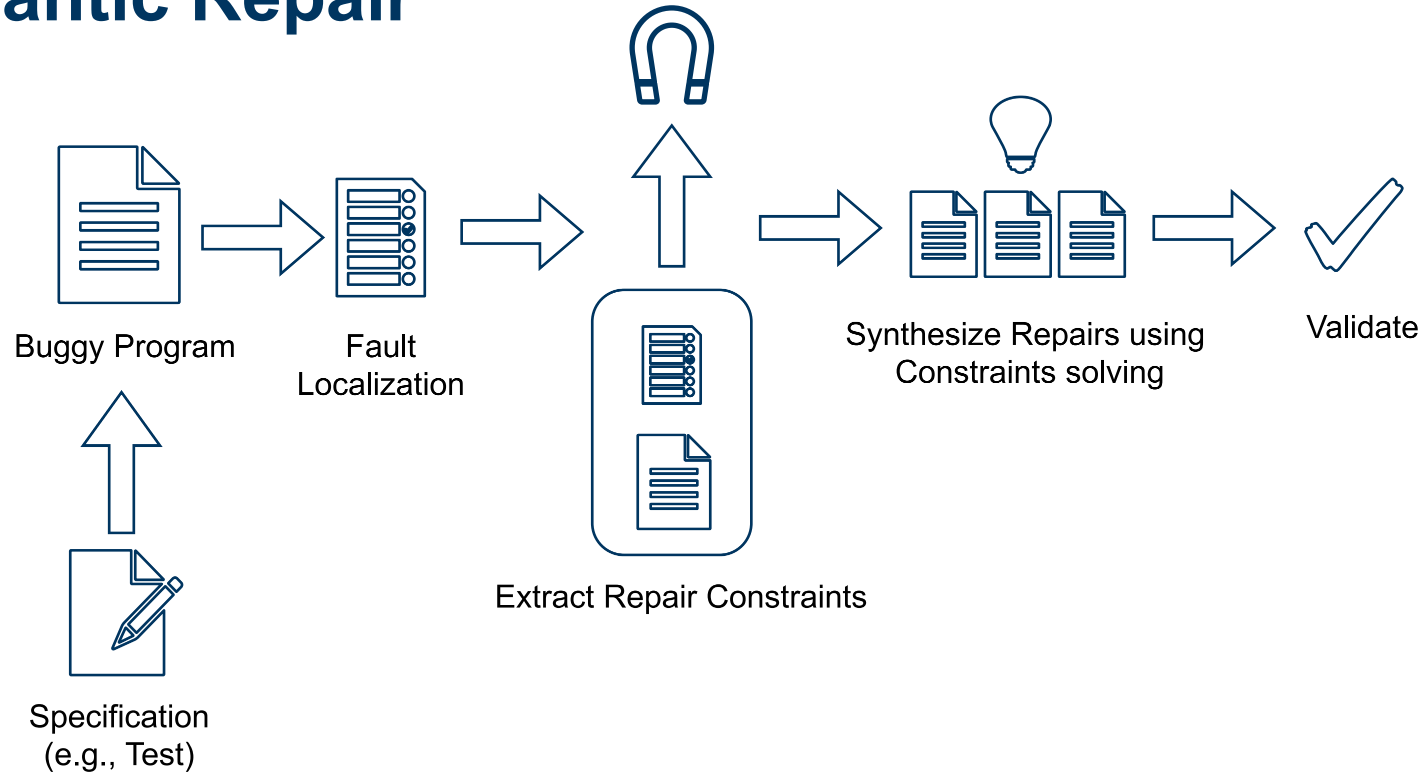
Since program repair needs to be guided by certain notions of correctness and formal specifications of the program's behavior are usually not available, it is common to use test-suites to guide repair. The goal of automated repair is then to produce a (minimal) modification of the program so as to pass the tests in the given test-suite. While test-suite driven repair provides a practical formulation of the program repair problem, it gives rise to the phenomenon of "overfitting" [26, 30]. The patched program may pass the tests in the given test-suite while failing tests outside the test-suite, thereby overfitting the test data. Such overfitting patches are called *plausible* patches because they repair the failing test case(s), but they are not guaranteed to be *correct*, since they may fail tests outside the test-suite guiding the repair. Various solutions to alleviate the patch overfitting issue have been studied to date, including symbolic specification inference [23, 25], machine learning-based prioritization of patches [2, 20, 21] and fuzzing based test-suite augmentation [7]. These works do not guarantee any notion of correctness of the patches, and cannot guarantee even the most basic correctness criteria such as crash freedom.

In this work, we reflect on the problem of *patch overfitting* [22, 26, 30], in our attempt to produce patches which work

Semantic Repair (Constraint-based Repair)

- Construct a **repair constraint** that a program should satisfy
- Repair problem as a **synthesis** problem
- Use semantic approaches, e.g. **symbolic execution**, to extract the properties for the function to be synthesized
- Synthesize the program that **satisfies** the repair constraints/program properties

Semantic Repair



An Example

```
int length, index = 0;
int height[10], breadth[8];
input(length);
while (index < length) {
  height[index] = index + 1;
  ++index;
}
```

Input → length = 11

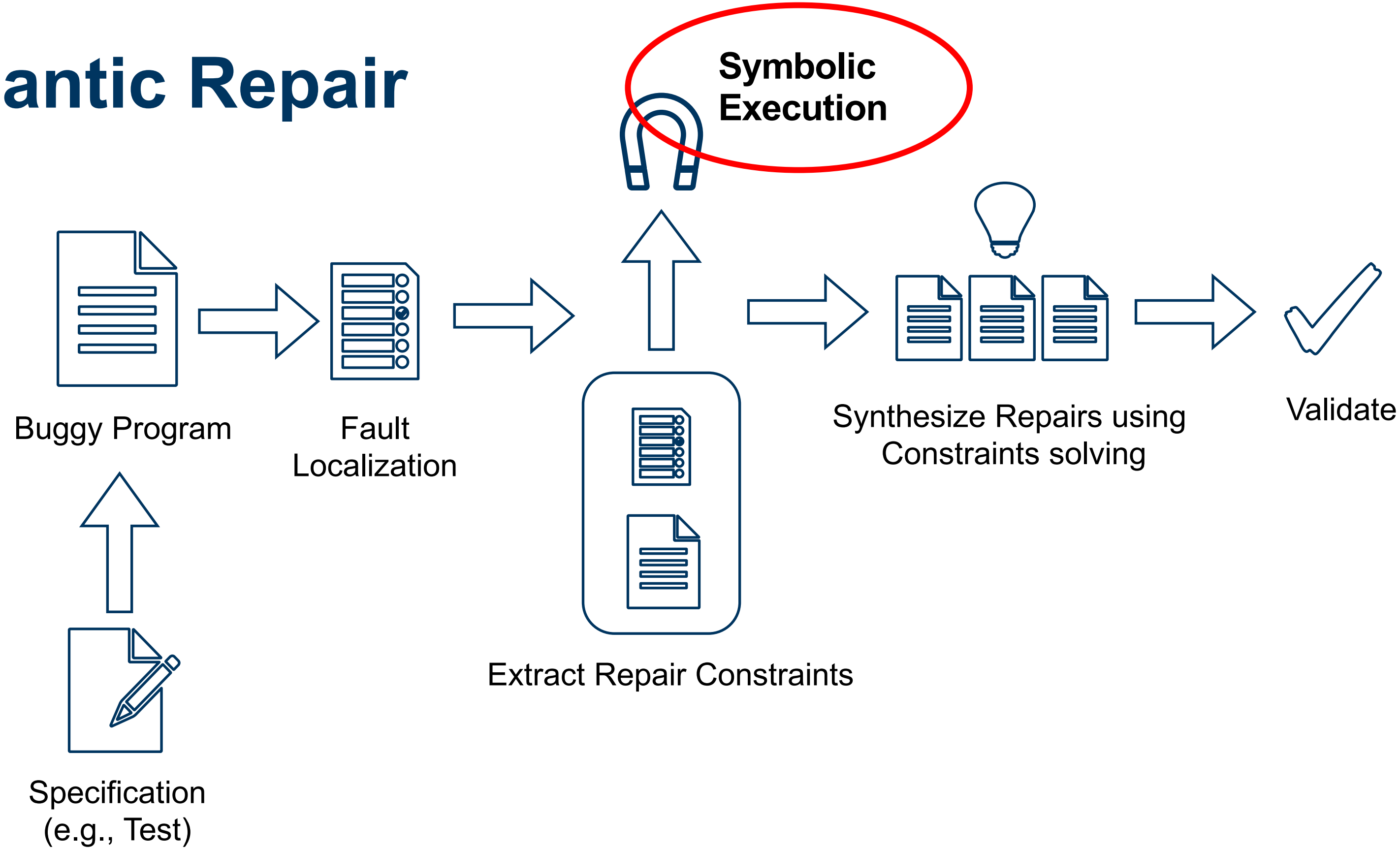
Constraint: index < sizeof(buff)

⇩

while (index < length & index < sizeof(height)) {

One potential repair.
Can generate more based on generated constraints.

Semantic Repair



Symbolic Execution

- introduced by King[1] and Clarke[2]
- analysis of programs with **unspecified inputs**, i.e. execute a program with **symbolic** inputs
- **symbolic states** represent sets of concrete states
- for each path, build a **path condition**
  - condition on inputs – for the execution to follow that path
  - check path condition satisfiability – explore only feasible paths
- symbolic state
  - symbolic values / expressions for variables
  - path condition
  - instruction pointer

[1] James C. King. 1976. Symbolic execution and program testing. Commun. ACM 19, 7 (July 1976), 385-394.
[2] L. A. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," in IEEE Transactions on Software Engineering, vol. SE-2, no. 3, pp. 215-222, Sept. 1976.

Example: concrete execution

code that swaps 2 integers

```
int x, y;
if (x > y) {
  x = x + y;
  y = x - y;
  x = x - y;
  if (x > y)
    assert false;
}
```

concrete execution path

x = 1, y = 0

x > y ? true

x = 1 + 0 = 1

y = 1 – 0 = 1

x = 1 – 1 = 0

0 > 1 ? false

END

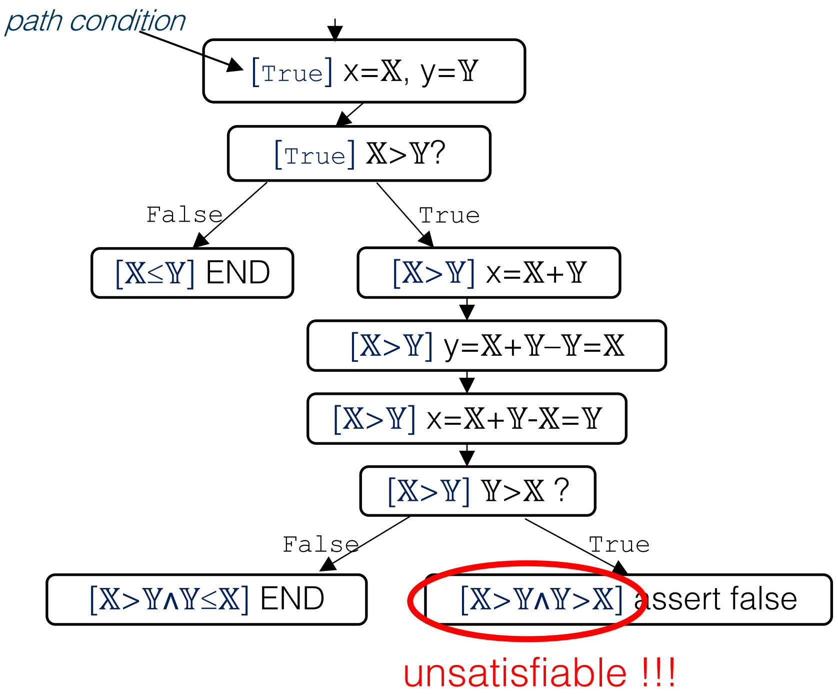
Example: symbolic execution

code that swaps 2 integers

```
int x, y;
if (x > y) {
  x = x + y;
  y = x - y;
  x = x - y;
  if (x > y)
    assert false;
}
```

Hint: solve PCs to obtain test inputs

symbolic execution tree



Decision Procedures

- Used to **check path conditions**
  - if path condition is unsatisfiable, backtrack
  - solutions of statisfiable constraints used as test inputs
- SMT solvers
  - **Satisfiability Modulo Theories**
  - Given a formula first-order logic, with associated background theories, is the formula satisfiable?
- See also:
  - SMTLIB – library for SMT formulas (common format) and tools
  - SMTCOMP – annual competition of SMT solvers
  - Z3 - https://rise4fun.com/z3/tutorial

Symbolic Execution: Limitations

- **Path explosion**
  - symbolically executing all program path does **not** scale well!
- Memory aliasing
  - accessing **same** memory with **difference** aliases
- Arrays
  - Array access with **symbolic indexes** are difficult to manage

SemFix: Program Repair via Semantic Analysis

- APR technique based on **symbolic execution**, **constraints solving**, and **program synthesis**
- Given a set of test cases
  - requirement for the repair is formulated as a **constraint**
  - **solve** the formulated **constraint** by iterating over a space of repair expressions

SemFix: Program Repair via Semantic Analysis

Hoang Duong Thien Nguyen Dawei Qi Abhik Roychoudhury
School of Computing, National University of Singapore, Singapore
{hoangdtn,dawei,abhik}@comp.nus.edu.sg

Satish Chandra
IBM Research, USA
satishchandra@us.ibm.com

Abstract—**Debugging consumes significant time and effort in any major software development project. Moreover, even after the root cause of a bug is identified, fixing the bug is non-trivial. Given this situation, automated program repair methods are of value. In this paper, we present an automated repair method based on symbolic execution, constraint solving and program synthesis. In our approach, the requirement on the repaired code to pass a given set of tests is formulated as a constraint. Such a constraint is then solved by iterating over a layered space of repair expressions, layered by the complexity of the repair code. We compare our method with recently proposed genetic programming based repair on SIR programs with seeded bugs, as well as fragments of GNU Coreutils with real bugs. On these subjects, our approach reports a higher success-rate than genetic programming based repair, and produces a repair faster.**

I. INTRODUCTION

Bug fixing continues to be a mostly manual, time consuming, and therefore expensive activity in software development. Therefore, automated techniques to repair buggy programs can be of tremendous value. In particular, given that compute cycles are cheap and abundant, it makes sense to investigate techniques that help shift the "heavy lifting" of program repair from the human to the computer. While a programmer might not blindly trust a computer-generated fix to her code, her task can become considerably easier: rather than figure out a fix, just verify that an automatically generated fix is correct. Not surprisingly, researchers have recently started looking into automated program repair tools [1]–[3].

We focus on general purpose programs, for which a test suite is available as a way to tell whether the program is working correctly (i.e. it passes all the tests) or not (i.e. there exists a failing test), but otherwise no formal specification of correct behavior is available; this is generally the case in practice (by contrast, kernels that manipulate data structures often do have specifications, and automatic repair on data structure programs have been well studied, for example see [4], [5]). A successful repair would be a modification of the program such that it passes all the tests in the test suite.

One of the most successful techniques in recent work that works on general programs is based on syntactic search. The premise behind this technique is that, once we know where the defective expression is in the program, a correct expression may be present syntactically at another place in the program, so it is a matter of searching over a space of replacements from among existing expressions.[1] The technique uses genetic programming technique for searching over this space, and has been shown to work for large programs [6]. The limitation of this technique is that the correct expression should be present in the program; the technique cannot "synthesize" an appropriate expression from variables and constants.

An obvious response to the limitation would be a search over a space of syntactic expressions, without consideration of whether those expressions appear elsewhere in the program. Such an approach would be more in the flavor of *sketching* [7], [8]. However, unless the space of repair expression is fixed upfront (possibly as a set of templates), such a technique will not work. Furthermore, as our experiments show, enumerating over the set of possible repair templates is inefficient.

In this paper, we explore a constraint based semantic approach towards program repair. The repair constraints are generated by our desire to have the repaired program pass the given test cases. Thus, given a program location to be fixed, we derive constraints on the expression to appear in the program location, in order to have the changed program pass all the given tests. The repair constraints are generated via (controlled) symbolic execution and the expression to be repaired is obtained via program synthesis. We report that, for certain kinds of program and bugs, the semantics-based approach can not only have a higher-success rate than a syntactic search-based approach, but also be able to produce a repair faster. At the same time, we do believe that symbolic execution imposes certain scalability limitations on the size of programs we can handle.

Our approach is a combination of three existing techniques.

- *Fault isolation*, i.e. where to fix the problem. The technique uses the ranking produced by a statistical fault isolation [9] tool (it shares this step with the search-based techniques.) Our approach examines *one buggy statement at a time* from a ranked suspicion report of statements.
- *Statement-level specification inference*. We automatically discover the correct specification of the buggy statement. We use an idea similar to the one used in angelic debugging [10] in converting an expression to a non-deterministic expression. This step allows us to create, for each input to the buggy statement, the output that would have resulted in the test passing.
- *Program synthesis*. The third idea is to use component-based synthesis idea [11] to synthesize an expression that conforms to the specification discovered before.

The inter-play of the second and third steps is the primary novelty of our repair tool. The statement-level specification narrows the search space significantly, and sets up the problem

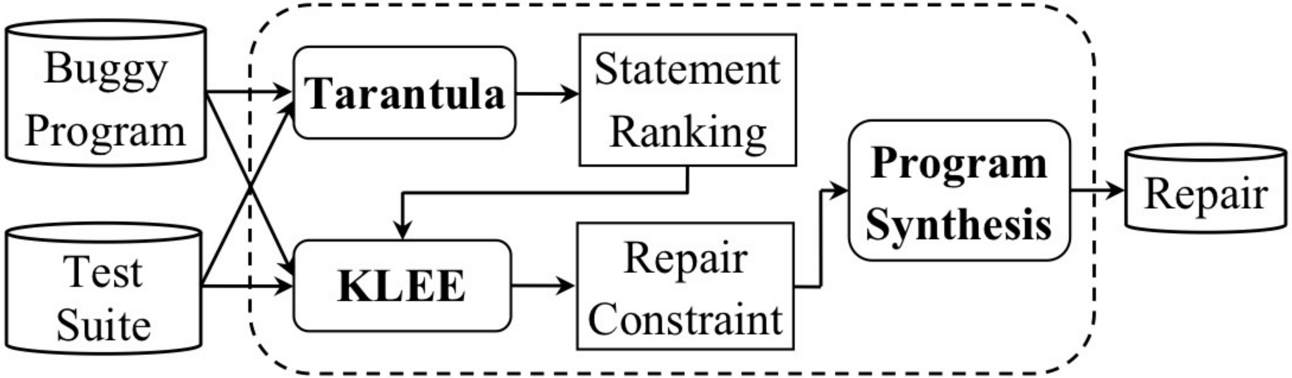
[1]This is an oversimplification, but broadly speaking this is the idea.

978-1-4673-3076-3/13/$31.00 © 2013 IEEE 772 ICSE 2013, San Francisco, CA, USA

https://doi.org/10.1109/ICSE.2013.6606623

RUHR UNIVERSITÄT BOCHUM RUB

Workflow of SemFix



KLEE is a symbolic execution engine built on top of the LLVM Compiler infrastructure:
https://klee.github.io

Example

Code excerpt from Tcas (Traffic collision avoidance system)

```
int is_upward_preferred (int inhibit, int up_sep, int down_sep) {
int bias;
if (inhibit)
    bias = down_sep; //fix: bias=up_sep+100
else
    bias = up_sep;
if (bias > down_sep)
return 1;
else
return 0:
```

Test	Inputs			Expected output	Observed output	Status
	inhibit	up_sep	down_sep			
1	1	0	100	0	0	pass
2	1	11	110	1	0	fail
3	0	100	50	1	1	pass
4	1	-20	60	1	0	fail
5	0	0	10	0	0	pass

Test Suite observing the fault

Fault Localization (using Tarantula)

Code excerpt from Tcas (Traffic collision avoidance system)

```
1. int is_upward_preferred (int inhibit, int up_sep, int down_sep) {
2.  int bias;
3.  if (inhibit)
4.      bias = down_sep; //fix: bias=up_sep+100
5. else
6.      bias = up_sep;
7.  if (bias > down_sep)
8. return 1;
9. else
10. return 0:
```

Test	Inputs			Expected output	Observed output	Status
	inhibit	up_sep	down_sep			
1	1	0	100	0	0	pass
2	1	11	110	1	0	fail
3	0	100	50	1	1	pass
4	1	-20	60	1	0	fail
5	0	0	10	0	0	pass

Test Suite observing the fault

Line	Score	Rank
4	0.75	1
10	0.6	2
3	0.5	3
7	0.5	3
6	0	5
8	0	5

Faulty Statements along with their rankings

Repair Synthesis and Symbolic Execution

Code excerpt from Tcas (Traffic collision avoidance system)

```
int is_upward_preferred (int inhibit, int up_sep, int down_sep) {
int bias;
if (inhibit)
    bias = down_sep; //fix: bias=up_sep+100
else
    bias = up_sep;
if (bias > down_sep)
return 1;
else
return 0;
```

Faulty Statement

```
bias = down_sep;
```

Repair Expression

```
bias = f(…);
```

Available vars

```
inhibit, up_sep,
down_sep, bias;
```

```
f(…);
```

```
f(int inhibit, int up_sep, int
down_sep, int bias);
```

```
f(int inhibit, int up_sep, int
down_sep);
```

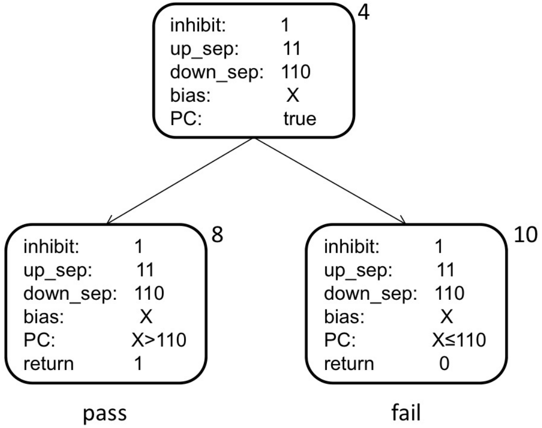
Uninitialized, thus non-usable

Repair Synthesis and Symbolic Execution

Repair Expression

```
bias = **f(int inhibit, int up_sep, int down_sep);**
```

find the constraint to be satisfied by f(…) to pass all test



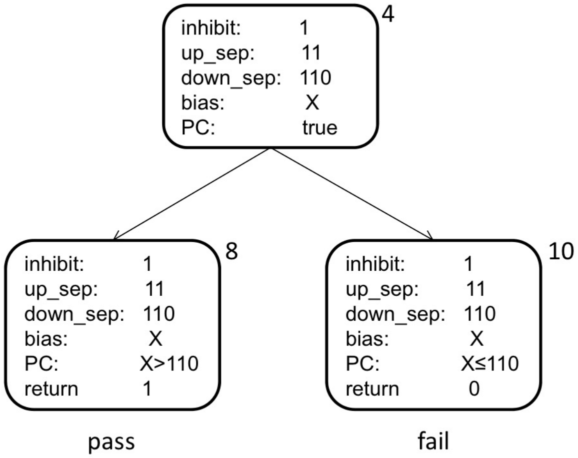
Symbolic execution based on Test 2

Test	Inputs			Expected output	Observed output	Status
	inhibit	up_sep	down_sep			
1	1	0	100	0	0	pass
2	1	11	110	1	0	fail
3	0	100	50	1	1	pass
4	1	-20	60	1	0	fail
5	0	0	10	0	0	pass

```
1. int is_upward_preferred (int inhibit, int up_sep, int down_sep)
   {
2. int bias;
3. if (inhibit)
4.     bias = down_sep; //fix: bias=up_sep+100
5. else
6.     bias = up_sep;
7. if (bias > down_sep)
8. return 1;
9. else
10. return 0;
```


RUHR UNIVERSITÄT BOCHUM RUB

Repair Synthesis and Symbolic Execution




```
1.  int is_upward_preferred (int inhibit, int up_sep, int down sep)
     {
2.  int bias;
3.  if (inhibit)
4.      bias = down_sep; //fix: bias=up_sep+100
5. else
6.      bias = up_sep;
7.  if (bias > down_sep)
8. return 1;
9. else
10. return 0;
```

$X > 110$

At line 4: inhibit == 1, up_sep = 11, down_sep = 110

bias = f(1, 11, 110); i.e., f(1, 11, 110) > 110

More constraints from given tests

f(1, 0, 110) <= 100 and f(1, -20, 60) > 60

Repair Synthesis and Symbolic Execution

Repair Constraint to satisfy

$$(f(1{,}11{,}110) > 110 \wedge f(1{,}0{,}100) \leq 100 \wedge f(1,-20{,}60) > 60)$$

$\vdots$

$$f(inhibit, up\_sep, down\_sep) = up\_sep + 100$$

$\vdots$

$$f(inhibit, up\_sep, down\_sep) = up\_sep - (-100)$$

Component-Based Program Synthesis

constants, +, -, …

Ingredients

SemFix: Highlights

- Generate repairs by modifying **only one statement**
- Generated repair **depends** on the given test suite
- Synthesize expression only on **the right hand side** of assignments/branch predicates
- The generated repair has one of the following two forms:
  - $x = f\_buggy\ (\ldots) \rightarrow x = f(\ldots)$
  - $if(f\_buggy) \rightarrow if(f(\ldots))$

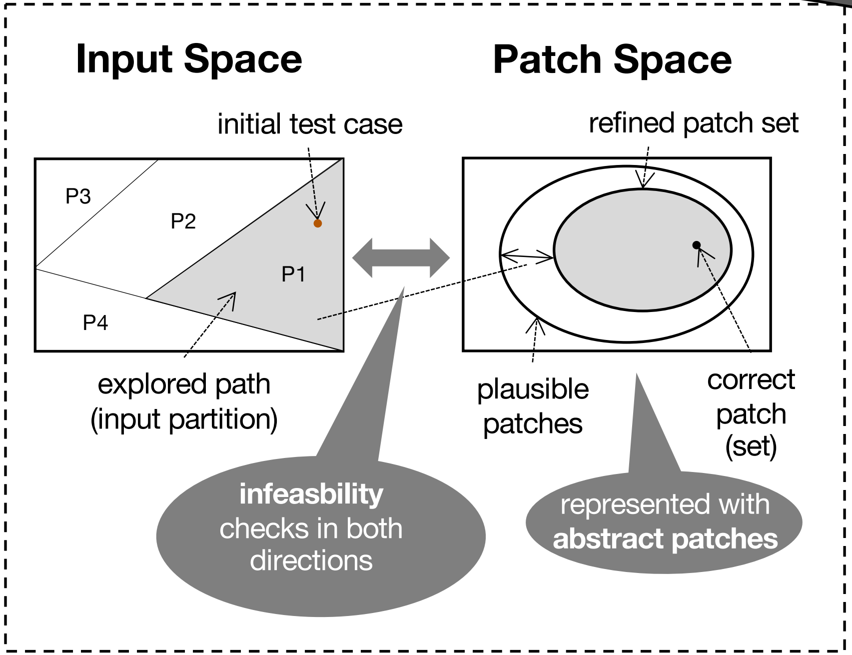
RUHR UNIVERSITÄT BOCHUM

Limitations

- Accuracy **decreases** with increasing number of tests
- **Depends** on test suite — Overfitting problem
- Single line repairs only

Concolic Program Repair

Fresh look on program repair



Concolic Program Repair

Ridwan Shariffdeen*
National University of Singapore
Singapore
ridwan@comp.nus.edu.sg

Yannic Noller*
National University of Singapore
Singapore
yannic.noller@acm.org

Lars Grunske
Humboldt-Universität zu Berlin
Germany
grunske@informatik.hu-berlin.de

Abhik Roychoudhury
National University of Singapore
Singapore
abhik@comp.nus.edu.sg

Abstract

Automated program repair reduces the manual effort in fixing program errors. However, existing repair techniques modify a buggy program such that it passes given tests. Such repair techniques do not discriminate between correct patches and patches that overfit the available tests (breaking untested but desired functionality). We propose an integrated approach for detecting and discarding overfitting patches via systematic co-exploration of the patch space and input space. We leverage concolic path exploration to systematically traverse the input space (and generate inputs), while ruling out significant parts of the patch space. Given a long enough time budget, this approach allows a significant reduction in the pool of patch candidates, as shown by our experiments. We implemented our technique in the form of a tool called 'CPR' and evaluated its efficacy in reducing the patch space by discarding overfitting patches from a pool of plausible patches. We evaluated our approach for fixing real-world software vulnerabilities and defects, for fixing functionality errors in programs drawn from SV-COMP benchmarks used in software verification, as well as for test-suite guided repair. In our experiments, we observed a patch space reduction due to our concolic exploration of up to 74% for fixing software vulnerabilities and up to 63% for SV-COMP programs. Our technique presents the viewpoint of *gradual correctness* — repair run over longer time leads to less overfitting fixes.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

*Joint first authors

PLDI '21, June 20–25, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8391-2/21/06...$15.00
https://doi.org/10.1145/3453483.3454051

Keywords: program repair, symbolic execution, program synthesis, patch overfitting

ACM Reference Format:
Ridwan Shariffdeen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. 2021. Concolic Program Repair. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3453483.3454051

1 Introduction

Automated Program Repair [14, 24] is an emerging technology which seeks to rectify errors or vulnerabilities in software automatically. There are various applications of automated repair, including improving programmer productivity, reducing exposure to software security vulnerabilities, producing self-healing software systems, and even enabling intelligent tutoring systems for teaching programming.

Since program repair needs to be guided by certain notions of correctness and formal specifications of the program's behavior are usually not available, it is common to use test-suites to guide repair. The goal of automated repair is then to produce a (minimal) modification of the program so as to pass the tests in the given test-suite. While test-suite driven repair provides a practical formulation of the program repair problem, it gives rise to the phenomenon of "overfitting" [26, 30]. The patched program may pass the tests in the given test-suite while failing tests outside the test-suite, thereby overfitting the test data. Such overfitting patches are called *plausible* patches because they repair the failing test case(s), but they are not guaranteed to be *correct*, since they may fail tests outside the test-suite guiding the repair. Various solutions to alleviate the patch overfitting issue have been studied to date, including symbolic specification inference [23, 25], machine learning-based prioritization of patches [2, 20, 21] and fuzzing based test-suite augmentation [7]. These works do not guarantee any notion of correctness of the patches, and cannot guarantee even the most basic correctness criteria such as crash freedom.

In this work, we reflect on the problem of *patch overfitting* [22, 26, 30], in our attempt to produce patches which work

https://doi.org/10.1145/3453483.3454051

RUHR UNIVERSITÄT BOCHUM

CPR: Inputs

Input

- Buggy Program
- Failing test case(s)
- Fix Locations
- User Specification

Buggy Program → CVE-2016-3623: Divide by Zero in LibTIFF v4.0.6

Failing test case(s) → e.g., exploit as TIFF picture

Fix Locations → source location, (fix template), synthesis components

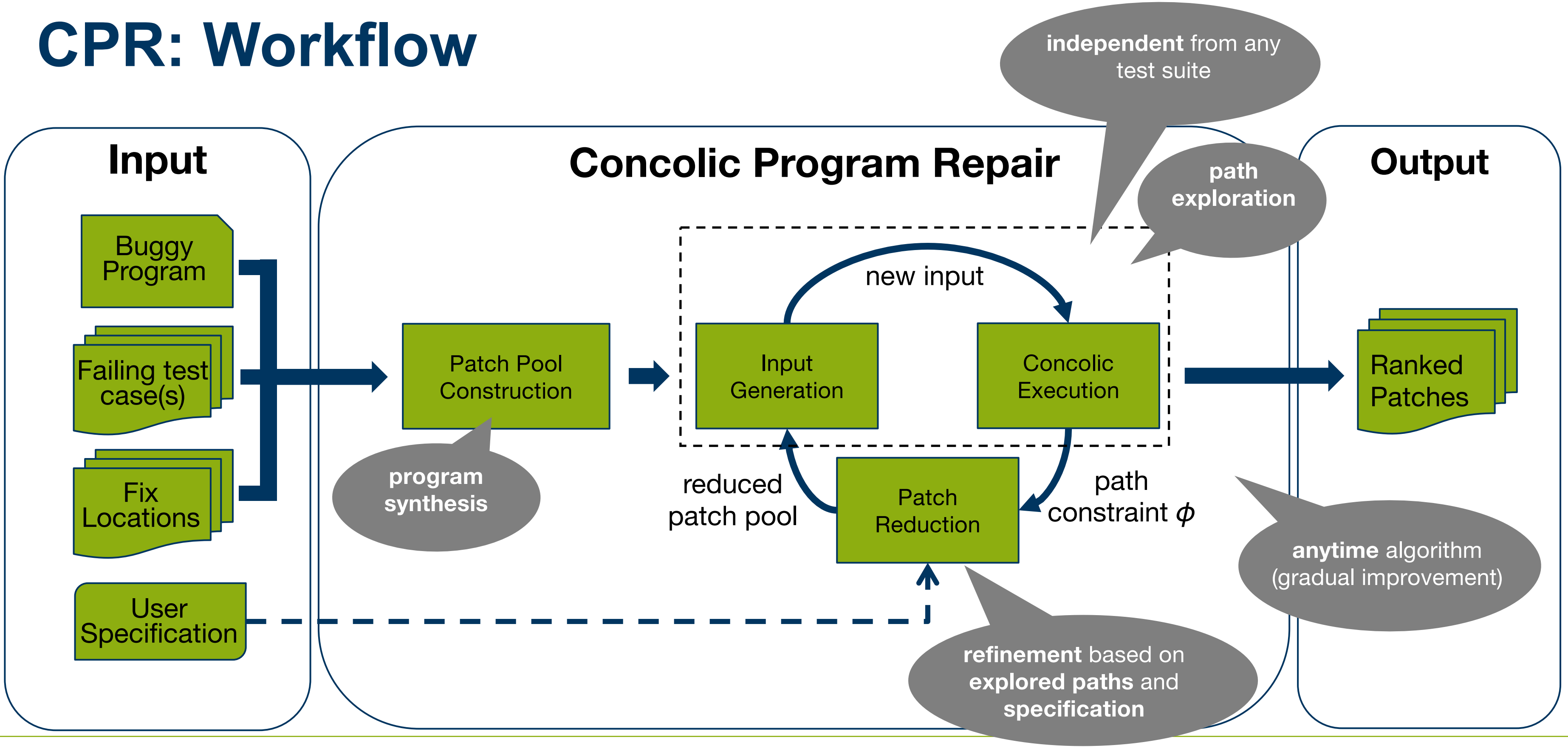
User Specification → formula about correct behavior in SMT format

```
........
static int
cvtRaster(TIFF* tif, uint32* raster, uint32 width, uint32 height)
{
    uint32 y;
    tstrip_t strip = 0;
    tsize_t cc, acc;
    unsigned char* buf;
    uint32 rwidth = roundup(width, horizSubSampling);
    uint32 rheight = roundup(height, vertSubSampling);
    uint32 nrows = (rowsperstrip > rheight ?
        rheight : rowsperstrip);
    uint32 rnrows = roundup(nrows,vertSubSampling);
    if (CONDITION) return 0;
    /* potential divide-by-zero error */
    cc = rnrows*rwidth + 2 * ((rnrows*rwidth)
        / (horizSubSampling*vertSubSampling));
    ........
}
```

observation

(assert (= false (= observation 0)))

CPR: Workflow



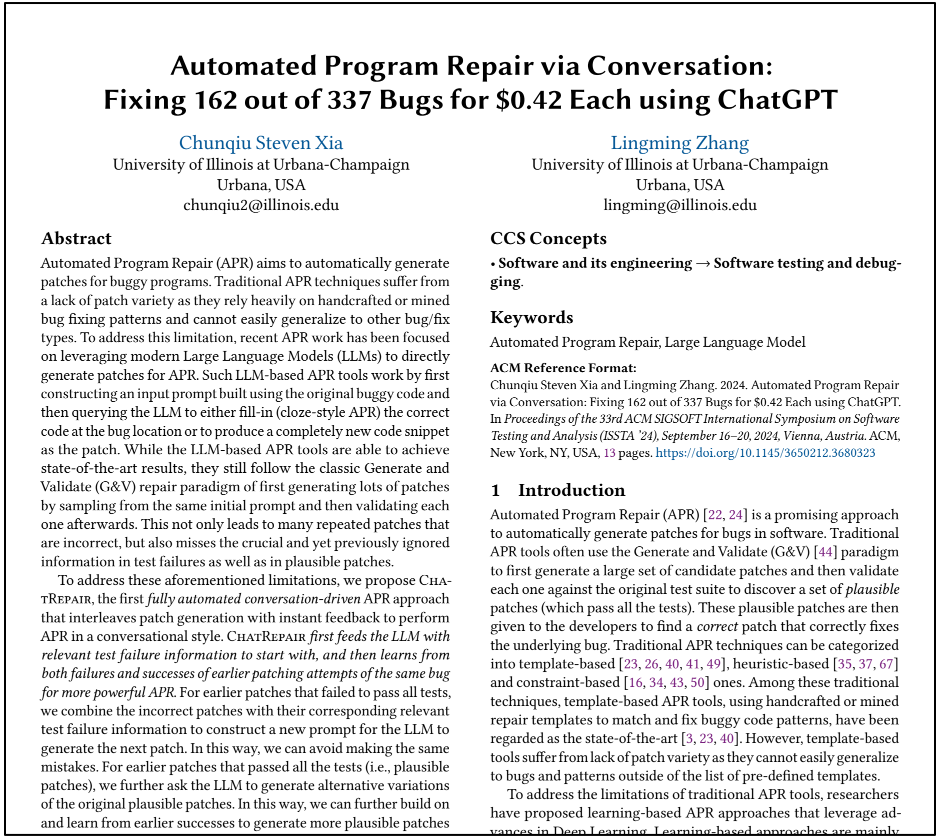
CPR: Conclusions

- Challenge 1: correctness
  - overfitting to test cases or scenarios without test cases
  - needs other types of specification, e.g., user-provided constraints
- Challenge 2: usability (integration into software development)
  - patch presentation → efficient ranking
  - efficient patch generation → rich and abstract patch space

Learning-based APR

Learning-based APR

- Many proposed approaches that **learn code transformations** from **code corpus**
  - Neural Machine Translation (NMT)
  - Sequence-to-Sequence Translation
- The learning based repair techniques **do not rely** on **pre-defined transformation** operators, enabling them to generate abundant kinds of patches by learning from **history patches**.
- In case of generating uncompilable or incorrect patches, the auto-generated patches by learning-based APR can also be validated using compilers and available test cases just like traditional APR techniques.
- However, the early learning-based APR also had a main limitation that they had been trained on **limited number of projects** and **hence only limited number of programming features**.



Automated Program Repair via Conversation: Fixing 162 out of 337 Bugs for $0.42 Each using ChatGPT

Chunqiu Steven Xia
University of Illinois at Urbana-Champaign
Urbana, USA
chunqiu2@illinois.edu

Lingming Zhang
University of Illinois at Urbana-Champaign
Urbana, USA
lingming@illinois.edu

Abstract

Automated Program Repair (APR) aims to automatically generate patches for buggy programs. Traditional APR techniques suffer from a lack of patch variety as they rely heavily on handcrafted or mined bug fixing patterns and cannot easily generalize to other bug/fix types. To address this limitation, recent APR work has been focused on leveraging modern Large Language Models (LLMs) to directly generate patches for APR. Such LLM-based APR tools work by first constructing an input prompt built using the original buggy code and then querying the LLM to either fill-in (cloze-style APR) the correct code at the bug location or to produce a completely new code snippet as the patch. While the LLM-based APR tools are able to achieve state-of-the-art results, they still follow the classic Generate and Validate (G&V) repair paradigm of first generating lots of patches by sampling from the same initial prompt and then validating each one afterwards. This not only leads to many repeated patches that are incorrect, but also misses the crucial and yet previously ignored information in test failures as well as in plausible patches.

To address these aforementioned limitations, we propose ChatRepair, the first *fully automated conversation-driven* APR approach that interleaves patch generation with instant feedback to perform APR in a conversational style. ChatRepair *first feeds the LLM with relevant test failure information to start with, and then learns from both failures and successes of earlier patching attempts of the same bug for more powerful APR.* For earlier patches that failed to pass all tests, we combine the incorrect patches with their corresponding relevant test failure information to construct a new prompt for the LLM to generate the next patch. In this way, we can avoid making the same mistakes. For earlier patches that passed all the tests (i.e., plausible patches), we further ask the LLM to generate alternative variations of the original plausible patches. In this way, we can further build on and learn from earlier successes to generate more plausible patches

CCS Concepts

• **Software and its engineering** → **Software testing and debugging.**

Keywords

Automated Program Repair, Large Language Model

ACM Reference Format:
Chunqiu Steven Xia and Lingming Zhang. 2024. Automated Program Repair via Conversation: Fixing 162 out of 337 Bugs for $0.42 Each using ChatGPT. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3680323

1 Introduction

Automated Program Repair (APR) [22, 24] is a promising approach to automatically generate patches for bugs in software. Traditional APR tools often use the Generate and Validate (G&V) [44] paradigm to first generate a large set of candidate patches and then validate each one against the original test suite to discover a set of *plausible* patches (which pass all the tests). These plausible patches are then given to the developers to find a *correct* patch that correctly fixes the underlying bug. Traditional APR techniques can be categorized into template-based [23, 26, 40, 41, 49], heuristic-based [35, 37, 67] and constraint-based [16, 34, 43, 50] ones. Among these traditional techniques, template-based APR tools, using handcrafted or mined repair templates to match and fix buggy code patterns, have been regarded as the state-of-the-art [3, 23, 40]. However, template-based tools suffer from lack of patch variety as they cannot easily generalize to bugs and patterns outside of the list of pre-defined templates.

To address the limitations of traditional APR tools, researchers have proposed learning-based APR approaches that leverage advances in Deep Learning. Learning-based approaches are mainly

https://doi.org/10.1145/3650212.3680323

- recent advances in Large Language Models (LLM), however, show very strong results!
- LLM as component that can generate patchess
- conversational repair to improve generated patches

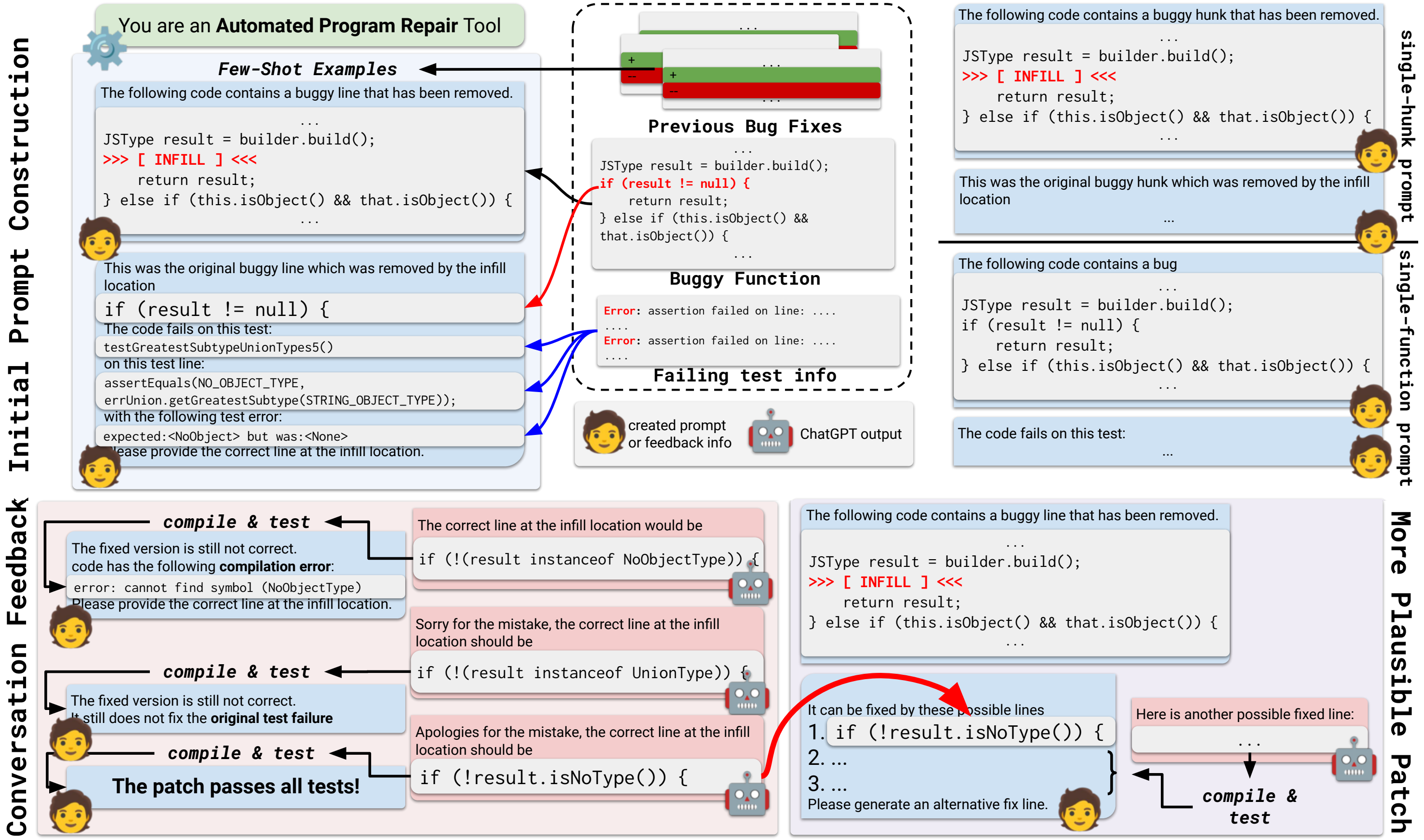

Initial Prompt Construction
You are an Automated Program Repair Tool
Few-Shot Examples
The following code contains a buggy line that has been removed.
...
JSType result = builder.build();
>>> [INFILL] <<<
return result;
} else if (this.isObject() && that.isObject()) {
...
This was the original buggy line which was removed by the infill location
if (result != null) {
The code fails on this test:
testGreatestSubtypeUnionTypes5()
on this test line:
assertEquals(NO_OBJECT_TYPE, errUnion.getGreatestSubtype(STRING_OBJECT_TYPE));
with the following test error:
expected:<NoObject> but was:<None>
Please provide the correct line at the infill location.
Previous Bug Fixes
Buggy Function
JSType result = builder.build();
if (result != null) {
return result;
} else if (this.isObject() &&
that.isObject()) {
Error: assertion failed on line: ....
Failing test info
created prompt or feedback info
ChatGPT output
The following code contains a buggy hunk that has been removed.
JSType result = builder.build();
>>> [INFILL] <<<
return result;
} else if (this.isObject() && that.isObject()) {
This was the original buggy hunk which was removed by the infill location
single-hunk prompt
The following code contains a bug
JSType result = builder.build();
if (result != null) {
return result;
} else if (this.isObject() && that.isObject()) {
The code fails on this test:
single-function prompt
Conversation Feedback
compile & test
The correct line at the infill location would be
if (!(result instanceof NoObjectType)) {
The fixed version is still not correct.
code has the following compilation error:
error: cannot find symbol (NoObjectType)
Please provide the correct line at the infill location.
Sorry for the mistake, the correct line at the infill location should be
if (!(result instanceof UnionType)) {
compile & test
The fixed version is still not correct.
It still does not fix the original test failure
Apologies for the mistake, the correct line at the infill location should be
if (!result.isNoType()) {
compile & test
The patch passes all tests!
More Plausible Patch
The following code contains a buggy line that has been removed.
JSType result = builder.build();
>>> [INFILL] <<<
return result;
} else if (this.isObject() && that.isObject()) {
It can be fixed by these possible lines
1. if (!result.isNoType()) {
2. ...
3. ...
Please generate an alternative fix line.
Here is another possible fixed line:
...
compile & test

Table 1: Correct fixes on Defects4j

Dataset	ChartRepair	BaseChatGPT	CodexRepair	FitRepair	AlphaRepair	SelfAPR	RewardRepair	Recoder	TBar	CURE
Chart	15	9	9	8	9	7	5	10	11	10
Closure	37	23	30	29	23	19	15	21	16	14
Lang	21	15	22	19	13	10	7	11	13	9
Math	32	25	29	24	21	22	19	18	22	19
Mockito	6	6	6	6	5	3	3	2	3	4
Time	3	2	3	3	3	3	1	3	3	1
D4J 1.2	**114**	80	99	89	74	64	50	65	68	57
D4J 2.0	**48**	25	31	44	36	31	25	11	8	-

Agentic Workflows

- Build a software engineering agent that can help with software maintenance!

AutoCodeRover: Autonomous Program Improvement

Yuntong Zhang
National University of Singapore
yuntong@comp.nus.edu.sg

Haifeng Ruan
National University of Singapore
hruan@comp.nus.edu.sg

Zhiyu Fan
National University of Singapore
zhiyufan@comp.nus.edu.sg

Abhik Roychoudhury
National University of Singapore
abhik@comp.nus.edu.sg

Abstract

Researchers have made significant progress in automating the software development process in the past decades. Automated techniques for issue summarization, bug reproduction, fault localization, and program repair have been built to ease the workload of developers. Recent progress in Large Language Models (LLMs) has significantly impacted the development process, where developers can use LLM-based programming assistants to achieve automated coding. Nevertheless, software engineering involves the process of program improvement apart from coding, specifically to enable software maintenance (e.g. program repair to fix bugs) and software evolution (e.g. feature additions). In this paper, we propose an automated approach for solving Github issues to autonomously achieve program improvement. In our approach called AutoCodeRover, LLMs are combined with sophisticated code search capabilities, ultimately leading to a program modification or patch. In contrast to recent LLM agent approaches from AI researchers and practitioners

CCS Concepts

• **Software and its engineering** → **Automatic programming**; **Maintaining software**; **Software testing and debugging**; • **Computing methodologies** → **Natural language processing**.

Keywords

large language model, automatic program repair, autonomous software engineering, autonomous software improvement

ACM Reference Format:

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3680384

1 Beyond Automatic Programming

Automating software engineering tasks has long been a vision

https://dl.acm.org/doi/pdf/10.1145/3650212.3680384

RepairAgent: An Autonomous, LLM-Based Agent for Program Repair

Islem Bouzenia
University of Stuttgart
Germany
fi_bouzenia@esi.dz

Premkumar Devanbu
UC Davis
USA
ptdevanbu@ucdavis.edu

Michael Pradel
University of Stuttgart
Germany
michael@binaervarianz.de

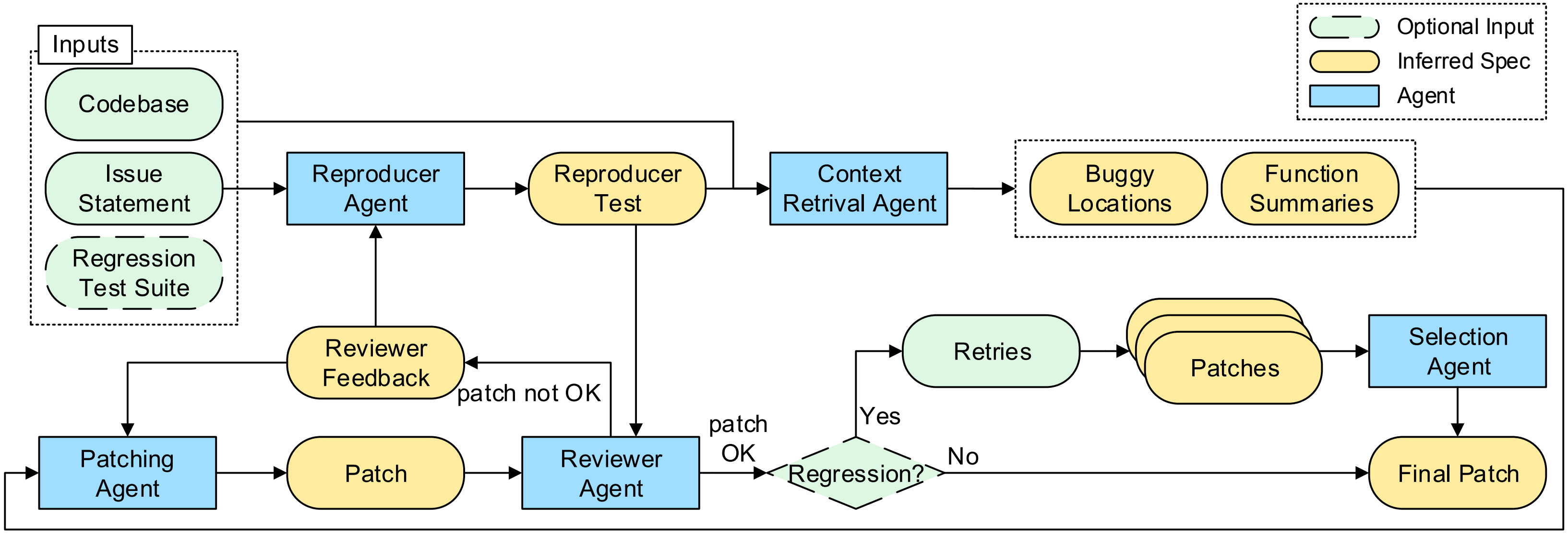
***Abstract*—Automated program repair has emerged as a powerful technique to mitigate the impact of software bugs on system reliability and user experience. This paper introduces RepairAgent, the first work to address the program repair challenge through an autonomous agent based on a large language model (LLM). Unlike existing deep learning-based approaches, which prompt a model with a fixed prompt or in a fixed feedback loop, our work treats the LLM as an agent capable of autonomously planning and executing actions to fix bugs by invoking suitable tools. RepairAgent freely interleaves gathering information about the bug, gathering repair ingredients, and validating fixes, while deciding which tools to invoke based on the gathered information and feedback from previous fix attempts. Key contributions that enable RepairAgent include a set of tools that are useful for program repair, a dynamically updated prompt format that allows the LLM to interact with these tools, and a finite state machine that guides the**

The current state-of-the-art in APR predominantly revolves around large language models (LLMs). The first generation of LLM-based repair uses a one-time interaction with the model, where the model receives a prompt containing the buggy code and produces a fixed version [17], [18]. The second and current generation of LLM-based repair introduces iterative approaches, which query the LLM repeatedly based on feedback obtained from previous fix attempts [19], [20], [21].

A key limitation of current iterative, LLM-based repair techniques is that their hard-coded feedback loops do not allow the model to gather information about the bug or existing code that may provide ingredients to fix the bug. Instead, these approaches fix the code context that is provided in the prompt, typically to the

https://arxiv.org/pdf/2403.17134

SpecRover



https://arxiv.org/pdf/2408.02232

Outlook on other topics

- Effective and Efficient patch validation
  - How to validate the correctness of the applied patch?
  - Will the patch introduce new problems?
  - Is the patch functionally correct?
- Trust in APR: what do the developers think?
- Other non-functional qualities, e.g., security and performance
- Patch Complexity (single-line, single-hunk/multi-line, multi-hunk)
- Static Analysis and APR, Fuzzing/Testing and APR
- Industry Applications: Facebook/Meta and Bloomberg (→ APR in the CI pipeline)
- APR in CS Education
- A central program repair website — https://program-repair.org

Summary

- Motivation for Automated Program Repair: Bugs! and the time to fix them!
- Components of APR
- Automated Fault Localization
- Types of Automated Program Repair (APR)
  - Search-based (Generate and Validate)
  - Semantic-based
  - Learning-based
    - APR in the era of Large Language Models (LLM)
    - Agentic Workflows for APR