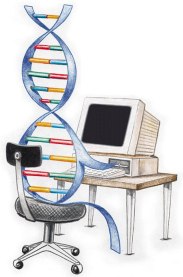
Carnegie Mellon University

isr institute for SOFTWARE RESEARCH

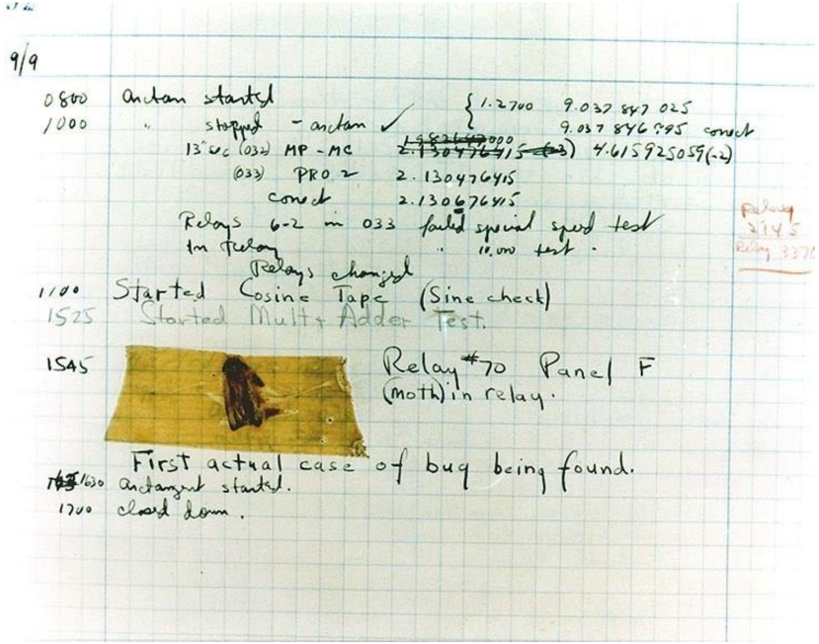
The Hitchhiker's Guide to

Search-Based Program Repair

Chris Timperley



1947

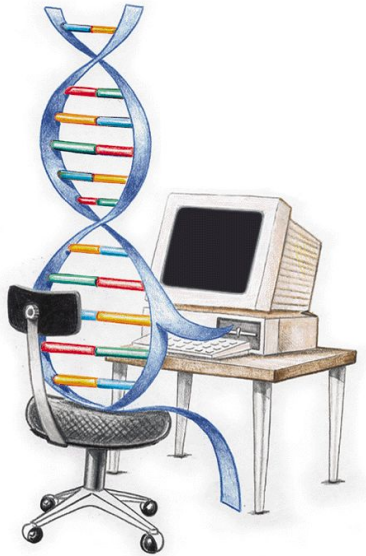

9/9
0800 Arctan started
1000 " stopped - arctan ✓
13" sec (032) MP - MC
(033) PRO 2 2.130476415
correct 2.130676415
Relays 6-2 in 033 failed special speed test
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Mult + Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
1630 Arctangent started.
1700 closed down.

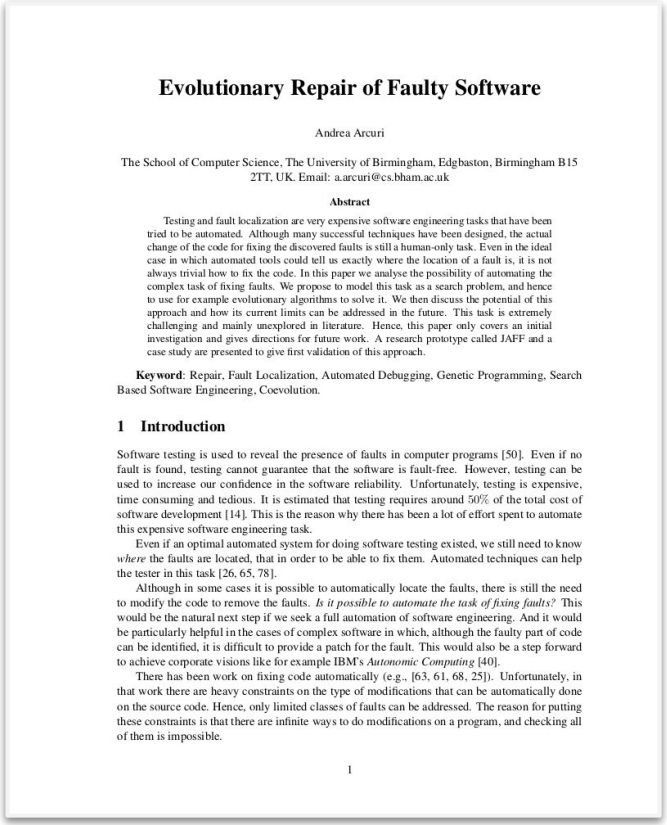
Ever since...

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.

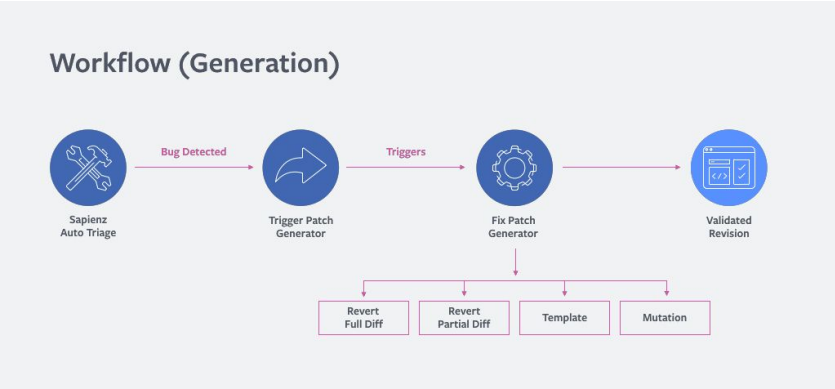
1

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

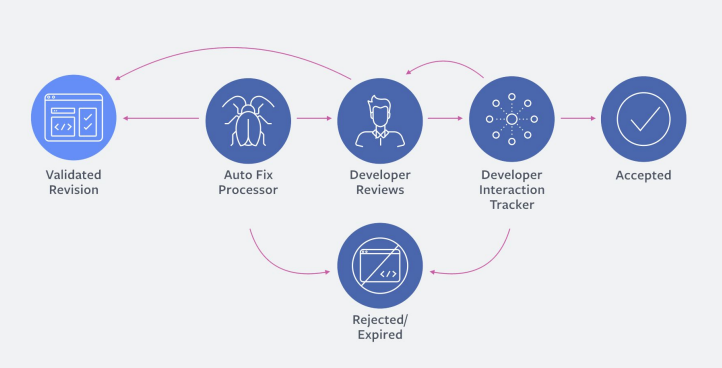
2009

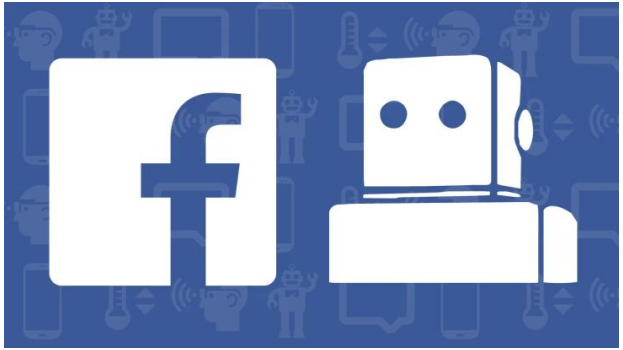
GenProg demonstrates program repair on real-world C programs

2018

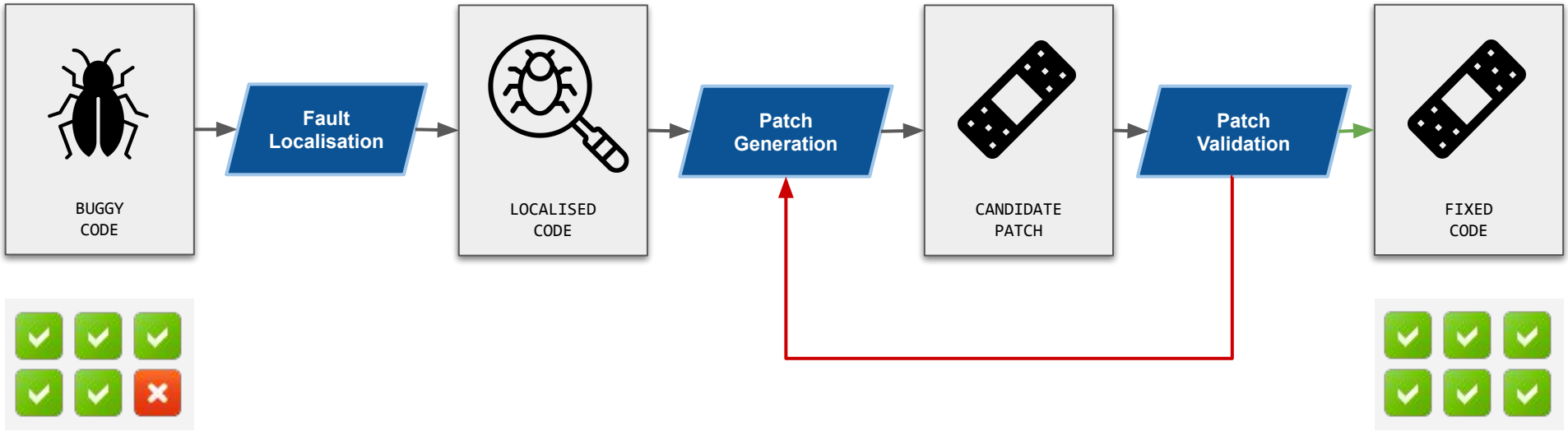


Facebook unveils SapFix: a search-based program repair technique





https://code.fb.com/developer-tools/finding-and-fixing-software-bugs-automatically-with-sapfix-and-sapienz/


BUGGY
CODE
Fault
Localisation
LOCALISED
CODE
Patch
Generation
CANDIDATE
PATCH
Patch
Validation
FIXED
CODE

Motivating Example

Support The Guardian
Available for everyone, funded by readers
Contribute
Subscribe
The Guardian
News Opinion Sport Culture Lifestyle More
Technology blog
Technology
Zune bricking code uncovered: it's a leap year mistake, and not Microsoft's
Editorially independent, open to everyone
We chose a different approach – will you support it?
Support The Guardian
most viewed in US
Baby T rex goes on sale on eBay, sparking paleontologists' outcry
Tiger Woods' Masters win was no tale of redemption - it was revenge
Marina Hyde
Deutsche Bank faces action over $20bn Russian money-laundering scheme
Cheating men's face shapes can give it away, study suggests
Strictly living room: 'breathtakingly horrific' Sydney real estate ad goes viral

TechHive
NEWS REVIEWS CONNECTED HOME ENTERTAINMENT DEALS RESOURCES
Home / MP3 Players
NEWS
Microsoft Says Leap Year Bug Caused Zune Failures
By Ginny Mies
PCWorld | DECEMBER 31, 2008 01:32 PM PT

Picture this: You're gearing up to create a killer playlist on your 30GB Zune for your annual New Year's bash. All of a sudden, your Zune locks up, reboots itself, and freezes. What the heck is going on?

In September, Microsoft released the Zune 3.0 software and firmware update along with its newest hardware, the Zune 120GB and 16GB models. At the time, Microsoft mentioned that it would be phasing out the older models slowly--but the company didn't say the change would be this dramatic.

As early as yesterday evening, reports of 30GB Zunes crashing began to surface on Microsoft support forums and gadget blogs. Microsoft updated the Zune support Web site with the following acknowledgement: "Customers with 30GB Zune devices may experience issues when booting their Zune hardware. We're aware of the problem and are working to correct it. Sorry for the inconvenience, and thanks for your patience!"

[Further reading: The best high-res digital audio players]

We contacted a Microsoft spokesperson, who confirmed the issue with this official statement:

"Early this morning we were alerted by our customers that there was a widespread issue affecting our 2006 model Zune 30GB devices (a large number of which are still actively being used). The technical team jumped on the problem immediately and isolated the issue: a bug in the internal clock driver related to the way the device handles a leap year.

"That being the case, the issue should be resolved over the next 24 hours as the time change moves to January 1, 2009. We expect the internal clock on the Zune 30GB devices will automatically reset tomorrow (noon, GMT). By tomorrow you should allow the battery to fully run out of power before the unit can restart successfully then simply ensure that your device is recharged, then turn it back on.

MORE LIKE THIS
ZuneGate '08: What Really Happened!
Microsoft launches Xbox Music: What you need to know
Zune services are finally, truly going away as Microsoft's Groove gets Sonos...
VIDEO
How to determine which free over-the-air TV channels you can tune in

http://www.theguardian.com/technology/blog/2009/jan/01/zune-firmware-mistake
http://www.techhive.com/article/156240/microsoft_zune_failure.html
https://www.silicon.co.uk/wp-content/uploads/2010/02/02/zunephone2.jpg
https://media.istockphoto.com/photos/common-house-brick-picture-id184374333?k=6&m=184374333&s=612x612&w=0&h=l8Q0QlBwFoJXxvh1zeeVXx56iznGPMitlzY5uAqy-Jw=

Can you spot the bug?

```
 1: int getYear(int days) {
 2:  int year = 1980;
 3:  while (days > 365) {
 4:   if (isLeapYear(year)) {
 5:    if (days > 366) {
 6:     days -= 366;
 7:     year += 1;
 8:    }
 9:   } else {
10:    days -= 365;
11:    year += 1;
12:   }
13:  }
14:  return year;
15: }
```

Determines the current year as a function of the number of days since 1st Jan 1980.

```
assert getYear(0) == 1980;
```


```
assert getYear(366) == 1980;
```


```
assert getYear(365) == 1980;
```


```
assert getYear(1827) == 1984;
```


```
assert getYear(367) == 1981;
```


```
assert getYear(1826) == 1984;
```


year(-366) == 1980

year(-100) == 1980

year(0) == 1980

year(365) == 1980

year(367) == 1981

year(1000) == 1982

year(1826) == 1984

year(2000) == 1985

Positive Tests

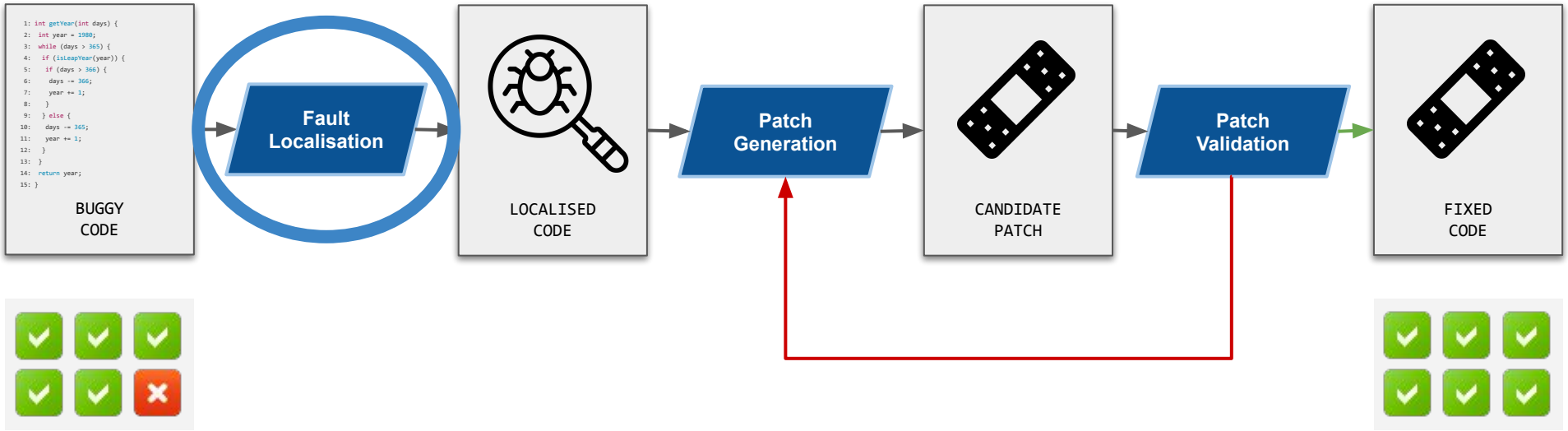
year(10593) == 2008

year(12054) == 2012

year(1827) == 1984

year(366) == 1980

Negative Tests


1: int getYear(int days) {
2: int year = 1980;
3: while (days > 365) {
4: if (isLeapYear(year)) {
5: if (days > 366) {
6: days -= 366;
7: year += 1;
8: }
9: } else {
10: days -= 365;
11: year += 1;
12: }
13: }
14: return year;
15: }
BUGGY CODE
Fault Localisation
LOCALISED CODE
Patch Generation
CANDIDATE PATCH
Patch Validation
FIXED CODE

Spectrum-Based Fault Localisation (SBFL)

Computes a measure of *suspiciousness* μ_c for each component *c* within the program (i.e., a line or statement), based on its correlation with positive and negative test execution.

- Components that are executed by passing tests are deemed less suspicious.
- Components that are executed by failing tests are deemed more suspicious.
- We restrict our attention to the set of *implicated* components:

$$\{c \in C \mid \mu_c > 0\}$$

SBFL (1/4): Coverage

```
int getYear(int days) {
  int year = 1980;
  while (days > 365) {
    if (isLeapYear(year)){
      if (days > 366) {
        days -= 366;
        year += 1;
      } else { }
    } else {
      days -= 365;
      year += 1;
    }
  }
  return year;
}
```

year(-366) == 1980

```
int getYear(int days) {
  int year = 1980;
  while (days > 365) {
    if (isLeapYear(year)){
      if (days > 366) {
        days -= 366;
        year += 1;
      } else { }
    } else {
      days -= 365;
      year += 1;
    }
  }
  return year;
}
```

year(367) == 1981

```
int getYear(int days) {
  int year = 1980;
  while (days > 365) {
    if (isLeapYear(year)){
      if (days > 366) {
        days -= 366;
        year += 1;
      } else { }
    } else {
      days -= 365;
      year += 1;
    }
  }
  return year;
}
```

year(1000) == 1982

```
int getYear(int days) {
  int year = 1980;
  while (days > 365) {
    if (isLeapYear(year)){
      if (days > 366) {
        days -= 366;
        year += 1;
      } else { }
    } else {
      days -= 365;
      year += 1;
    }
  }
  return year;
}
```

year(366) == FAIL

SBFL (2/4): Fault Spectra

		year(-366)	year(367)	year(1000)	year(366)
S1	`year = 1980;`	1	1	1	1
S2	`while (days > 365)`	1	1	1	1
S3	`if (isLeapYear(year))`	0	1	1	1
S4	`if (days > 366)`	0	1	1	1
S5	`days -= 366;`	0	1	1	0
S6	`year += 1;`	0	1	1	0
S7	`else { }`	0	0	0	1
S8	`else`	0	0	1	0
S9	`days -= 365;`	0	0	1	0
S10	`year += 1;`	0	0	1	0
S11	`return year;`	1	1	1	0
		PASS	*PASS*	*PASS*	*FAIL*

SBFL (3/4): Fault Spectra

		ep	ef	np	nf
S1	`year = 1980;`	3	1	0	0
S2	`while (days > 365)`	3	1	0	0
S3	`if (isLeapYear(year))`	2	1	1	0
S4	`if (days > 366)`	2	1	1	0
S5	`days -= 366;`	2	0	1	1
S6	`year += 1;`	2	0	1	1
S7	`else { }`	0	1	3	0
S8	`else`	1	0	0	1
S9	`days -= 365;`	1	0	0	1
S10	`year += 1;`	1	0	0	1
S11	`return year;`	3	0	0	1

e_p : num. passing tests that cover the line.
e_f : num. failing tests that cover the line.
n_p : num. passing tests that don't cover the line.
n_f : num. failing tests that don't cover the line.

SBFL (4/4): Suspiciousness

		ep	ef	np	nf
S1	`year = 1980;`	3	1	0	0
S2	`while (days > 365)`	3	1	0	0
S3	`if (isLeapYear(year))`	2	1	1	0
S4	`if (days > 366)`	2	1	1	0
S5	`days -= 366;`	2	0	1	1
S6	`year += 1;`	2	0	1	1
S7	`else { }`	0	1	3	0
S8	`else`	1	0	0	1
S9	`days -= 365;`	1	0	0	1
S10	`year += 1;`	1	0	0	1
S11	`return year;`	3	0	0	1

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

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

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

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

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

SBFL (4/4): Suspiciousness

		ep	ef	np	nf	μ
S1	`year = 1980;`	3	1	0	0	0.1
S2	`while (days > 365)`	3	1	0	0	0.1
S3	`if (isLeapYear(year))`	2	1	1	0	0.1
S4	`if (days > 366)`	2	1	1	0	0.1
S5	`days -= 366;`	2	0	1	1	0.0
S6	`year += 1;`	2	0	1	1	0.0
S7	`else { }`	0	1	3	0	1.0
S8	`else`	1	0	0	1	0.0
S9	`days -= 365;`	1	0	0	1	0.0
S10	`year += 1;`	1	0	0	1	0.0
S11	`return year;`	3	0	0	1	0.0

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

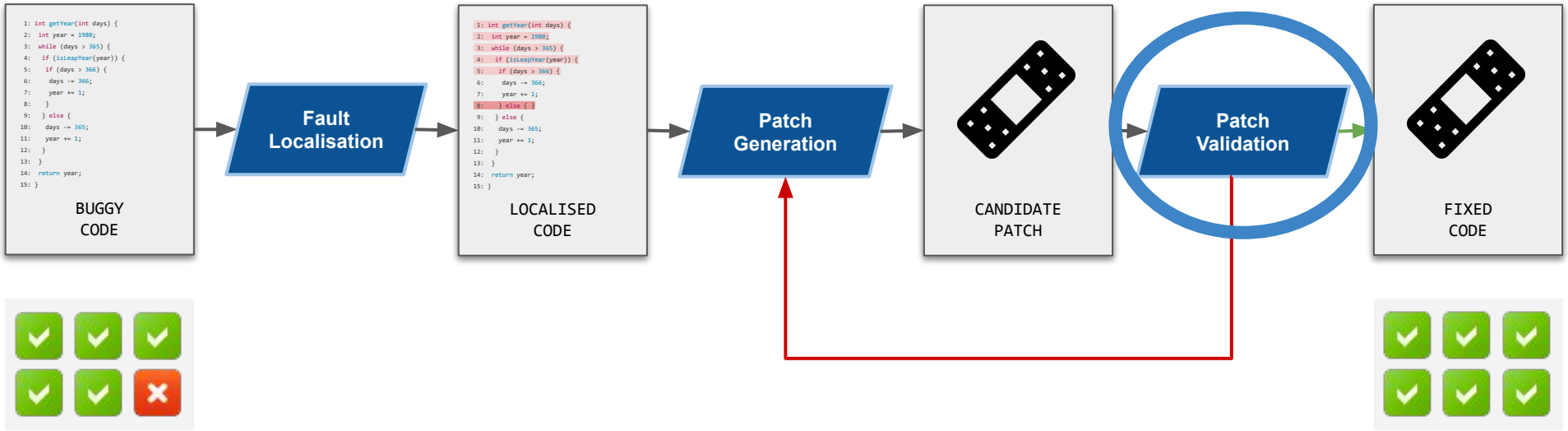
$$\frac{e_f}{\sqrt{(e_f + n_f) \cdot (e_f + e_p)}}$$

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

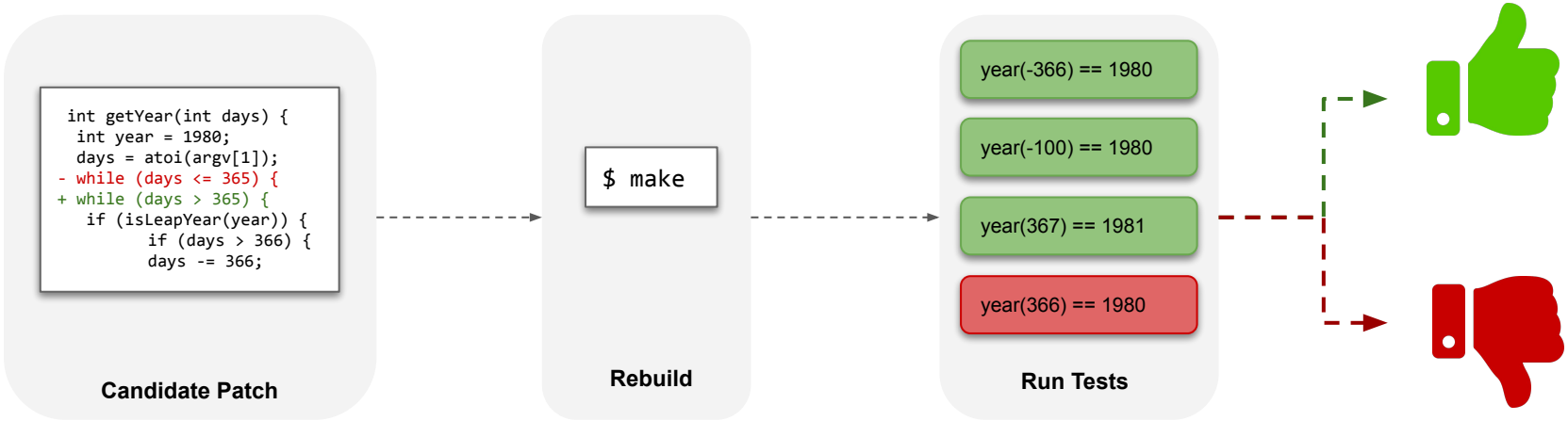
$$\frac{\frac{e_f}{e_f + n_f}}{\frac{e_p}{e_p + n_p} + \frac{e_f}{e_f + n_f}}$$

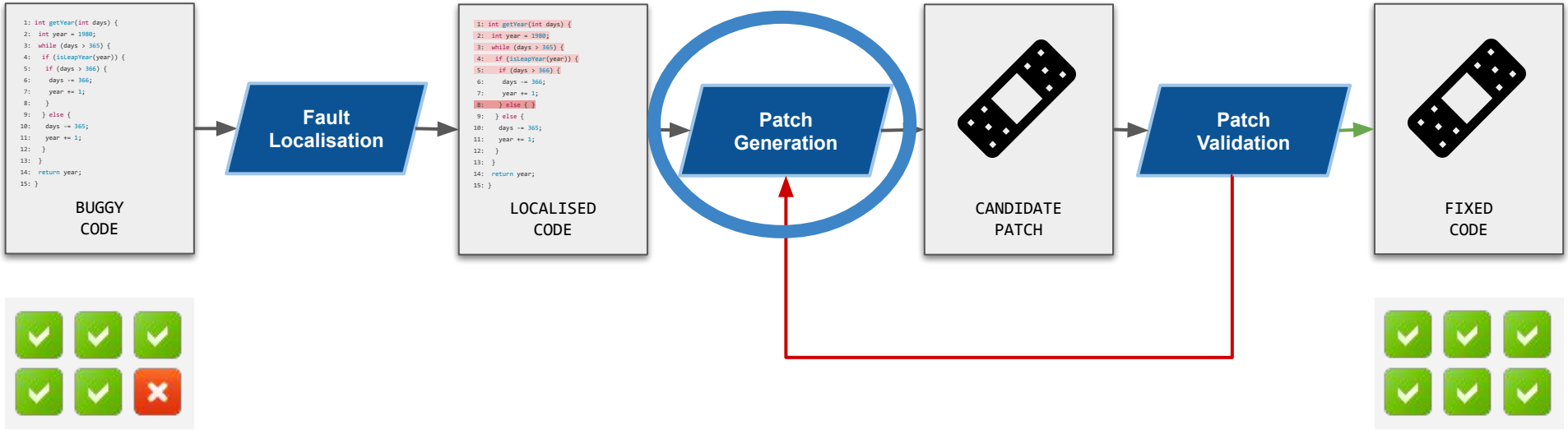
$$\begin{cases} 0, & \text{if } e_f + e_p = 0 \\ 1.0, & \text{if } e_f > 0 \wedge e_p = 0 \\ 0.1, & \text{otherwise} \end{cases}$$

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.). 244-258.


1: int getYear(int days) {
2: int year = 1980;
3: while (days > 365) {
4: if (isLeapYear(year)) {
5: if (days > 366) {
6: days -= 366;
7: year += 1;
8: }
9: } else {
10: days -= 365;
11: year += 1;
12: }
13: }
14: return year;
15: }
BUGGY CODE
Fault Localisation
LOCALISED CODE
Patch Generation
CANDIDATE PATCH
Patch Validation
FIXED CODE

Patch Validation




1: int getYear(int days) {
2: int year = 1980;
3: while (days > 365) {
4: if (isLeapYear(year)) {
5: if (days > 366) {
6: days -= 366;
7: year += 1;
8: }
9: } else {
10: days -= 365;
11: year += 1;
12: }
13: }
14: return year;
15: }
BUGGY CODE
Fault Localisation
1: int getYear(int days) {
2: int year = 1980;
3: while (days > 365) {
4: if (isLeapYear(year)) {
5: if (days > 366) {
6: days -= 366;
7: year += 1;
8: } else { }
9: } else {
10: days -= 365;
11: year += 1;
12: }
13: }
14: return year;
15: }
LOCALISED CODE
Patch Generation
CANDIDATE PATCH
Patch Validation
FIXED CODE

Patch Generation: Two Paradigms

Semantics-Based Repair
Angelix, SearchRepair, NOPOL, …

```
if (α0) {
  days -= α1;
  year += α2;
}
```

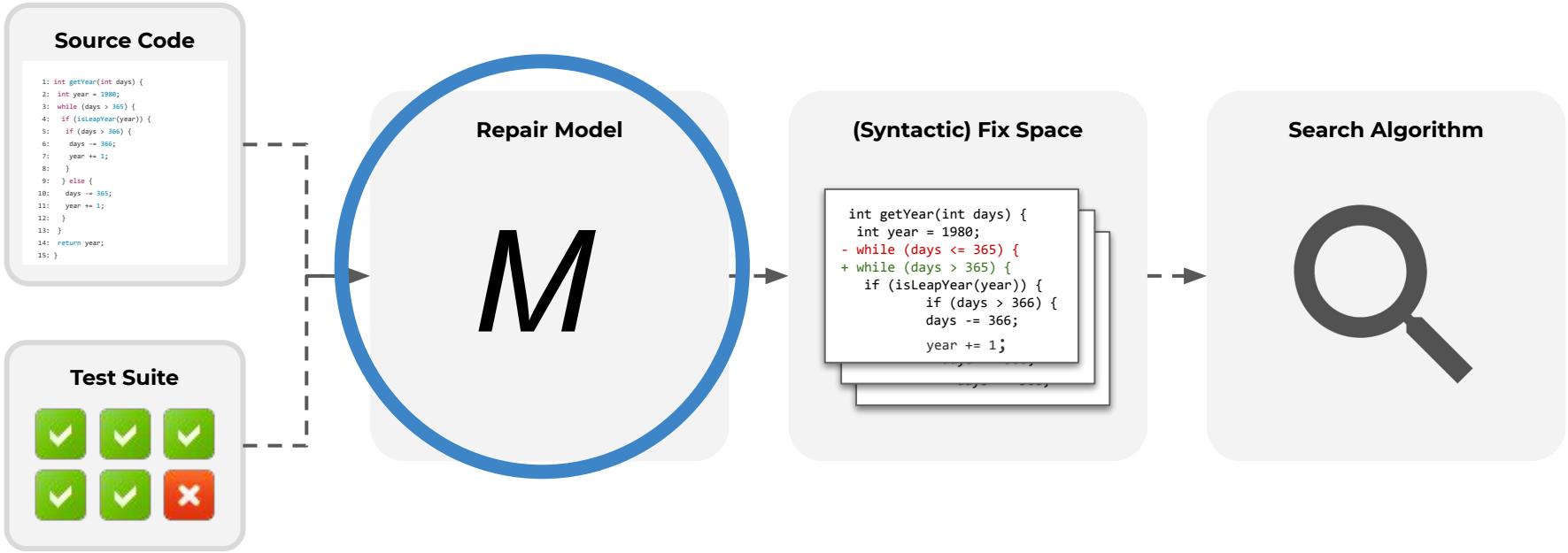
- Replaces suspicious *expressions* with symbolic variables (e.g., α_0).
- Uses concolic execution to find *angelic values* that would cause a given test to pass.
- Uses *program synthesis* to construct a replacement expression that produces the angelic values.
- Validate the candidate patch.

Syntax-Based Repair
GenProg, RSRepair, SPR, AE, PAR, …

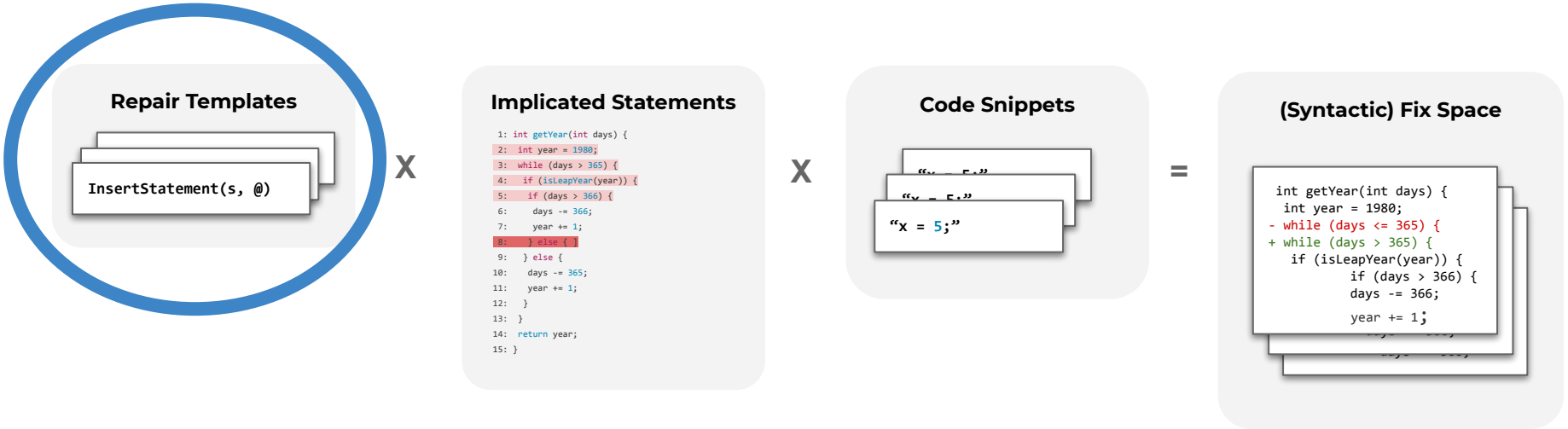
```
 int getYear(int days) {
  int year = 1980;
- while (days <= 365) {
+ while (days > 365) {
   if (isLeapYear(year)) {
         if (days > 366) {
```

- Applies repair operators to suspicious *statements* within the program to produce *concrete patches*.
- Validate candidate patch.
- Operates directly on source code: searches over the space of possible syntactic changes.

Syntax-Based Repair



Repair Model



Repair Model: Templates

Describes a transformation to the AST of the buggy program, which may or may not take a set of code snippets as its arguments.

Generic Templates

Highly expressive: can represent most changes.

- Delete Statement
- Replace Statement
- Append Statement
- Swap Statements
- Replace Expression
- Insert Conditional
- Insert Conditional Control Flow
- ...

Specialised Templates

Low expressiveness: tailored to particular bugs.

- Insert null check
- Insert bounds check
- Insert class cast check
- Apply off-by-one correction to array access
- Assign lower bound to array access
- Assign upper bound to array access
- Add, remove, or replace parameter in method call
- Replace method
- ...

Repair Model: GenProg

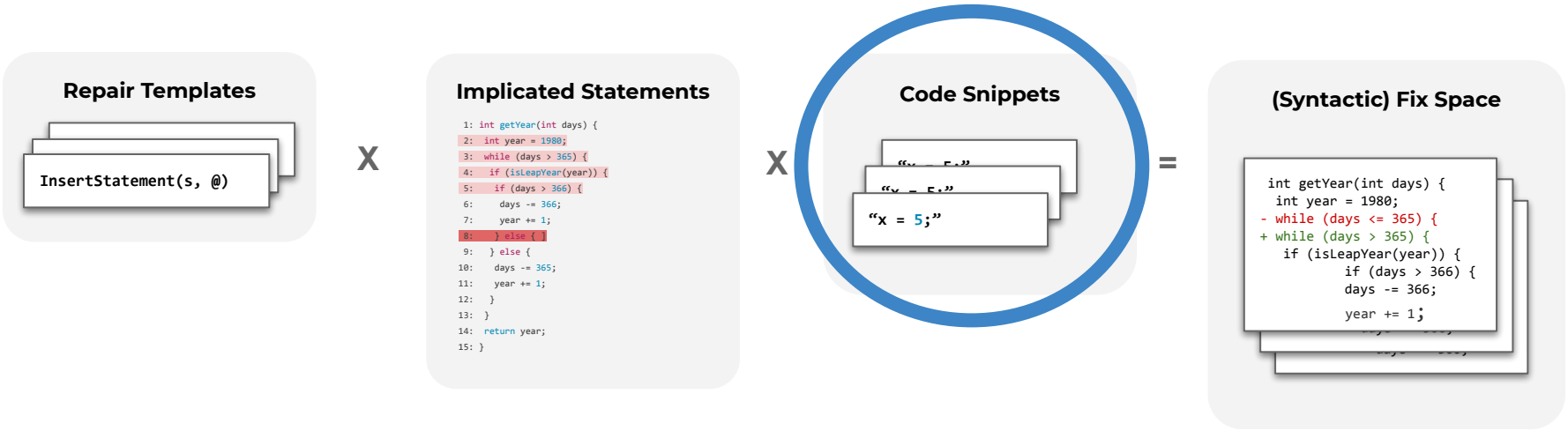
Uses a generic set of statement-level templates:

- Delete Statement
- Replace Statement
- Swap Statements
- Append Statement

In theory, this set of operators should be expressive enough to allow us to fix the vast majority of bugs.

Where do the statements come from?

Repair Model



Repair Model: Code Snippets

We need a source of code snippets to serve as template parameters.

- Searching the space of all possible statements is infeasible.
- We need a *heuristic* for selecting template parameter values.

Plastic Surgery Hypothesis

GenProg balances tractability and expressiveness by using code from the buggy program to compose its repairs. GenProg (and many other approaches) exploit what is now known as the plastic surgery hypothesis:

> "Changes to a codebase contain snippets that *already exist* in the codebase at the time of the change, and these snippets can be *efficiently found* and exploited." [1]

- Barr et al. measure line-level *graftability* of 15,273 commits across several large Java projects.
- 43% of changes are graftable from the exact version of software being changed.
- 30% of grafts could be found within the same file at which the human-written change occurred.

[1] Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. 2014. The plastic surgery hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (FSE 2014). ACM, New York, NY, USA, 306-317. DOI: https://doi.org/10.1145/2635868.2635898

Exercise 1: Form pairs and find repairs.

```
 1: int getYear(int days) {
 2:  int year = 1980;
 3:  while (days > 365) {
 4:   if (isLeapYear(year)) {
 5:    if (days > 366) {
 6:     days -= 366;
 7:     year += 1;
 8:    } else { }
 9:   } else {
10:    days -= 365;
11:    year += 1;
12:   }
13:  }
14:  return year;
15: }
```

Repair Templates:

- Delete Statement
- Replace Statement
- Append Statement

```
while (days > 365) { ... }
```

```
year = 1980;
```

```
days -= 366;
```

```
year += 1;
```

```
else { ... }
```

```
days -= 365;
```

```
return year;
```

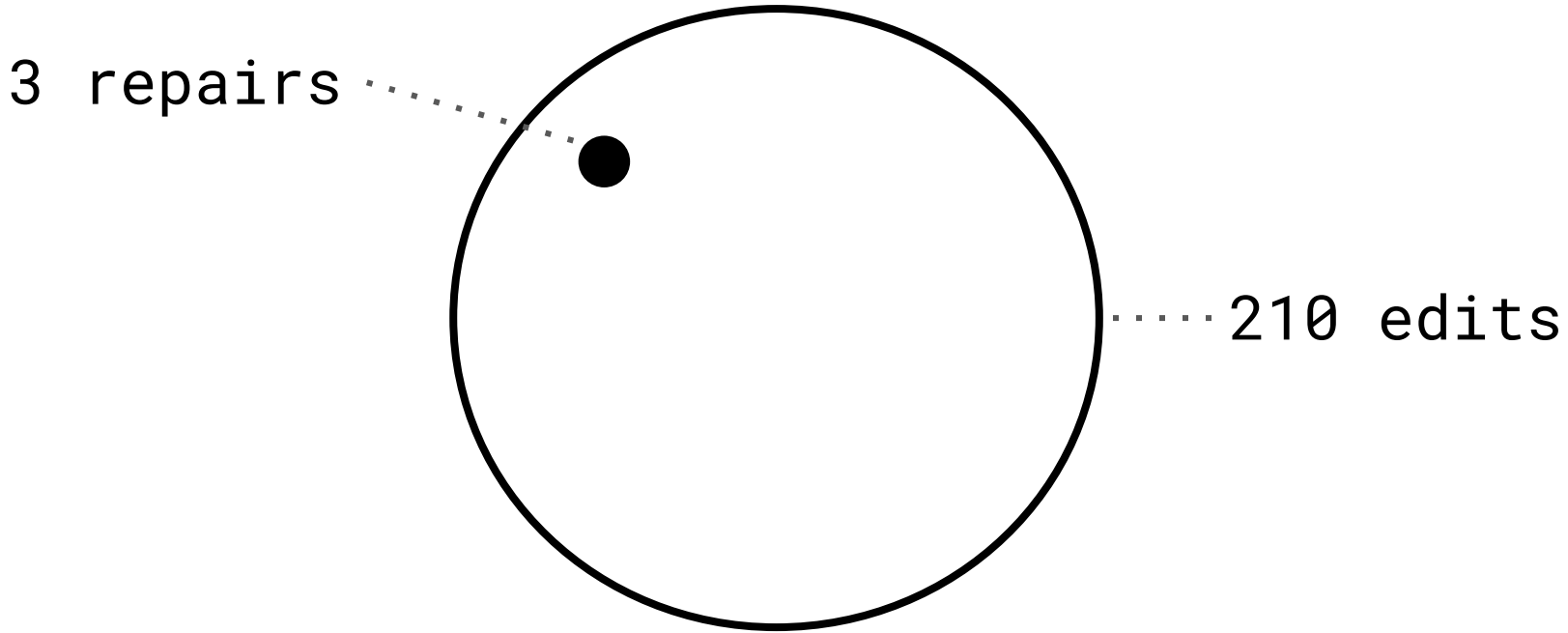
```
else {}
```

```
if (days > 366) { ... }
```

```
if (isLeapYear(year)) { ... } else {}
```

```
if (days <= 365) { break; }
```

```
break;
```



Likelihood of picking a repair: **~1.43%**

We can use *probabilistic models* to improve our odds of success.

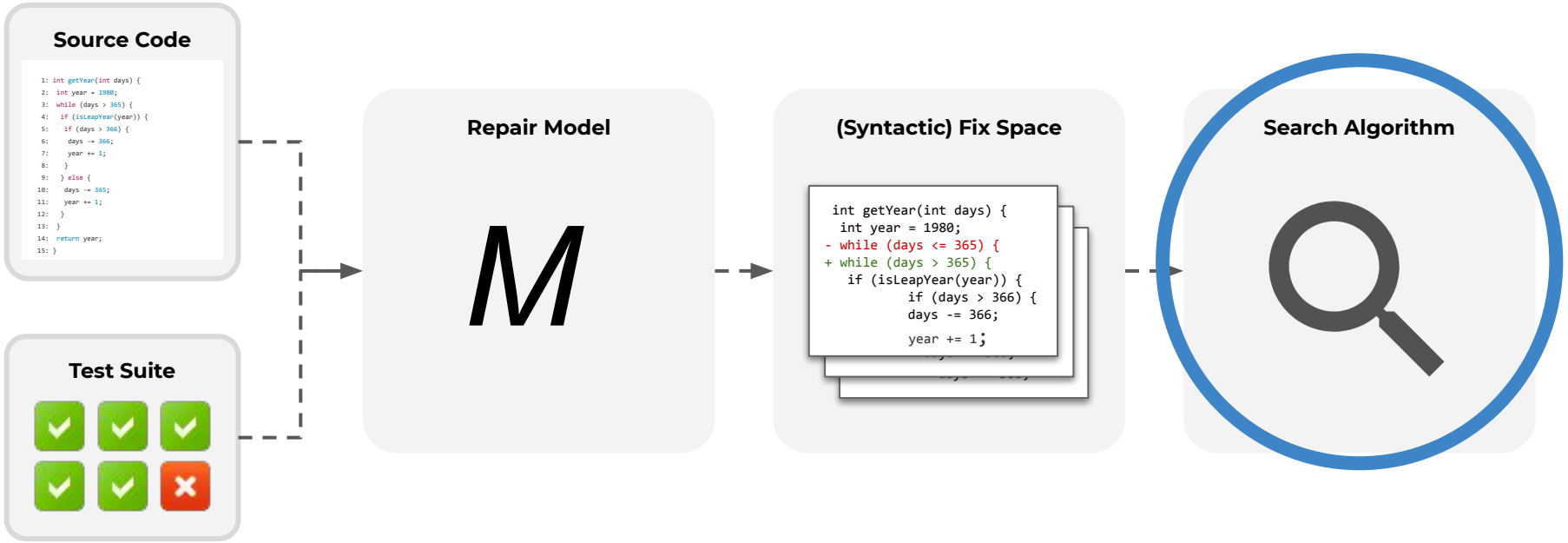
Probabilistic Repair Models

Learn a classifier, $p(e \mid \mathbf{x})$, that estimates the probability that an edit e is correct based on the features $\mathbf{x}$ of that edit:

- Edit Location (i.e., use fault localisation)
- Repair Template [1]
- Statement Kind [2]
- Syntactic Features [3]

[1] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015). ACM, New York, NY, USA, 166-178. DOI: https://doi.org/10.1145/2786805.2786811
[2] Mauricio Soto, Ferdian Thung, Chu-Pan Wong, Claire Le Goues, and David Lo. 2016. A deeper look into bug fixes: patterns, replacements, deletions, and additions. In Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16). ACM, New York, NY, USA, 512-515. DOI: https://doi.org/10.1145/2901739.2903495
[3] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16). ACM, New York, NY, USA, 298-312. DOI: https://doi.org/10.1145/2837614.2837617

Syntax-Based Repair



Random and Exhaustive Search

Use the probabilistic model to either (a) sample and evaluate edits at random [1], or (b) rank all possible edits and evaluate them in ranked order [2].

- Simple and surprisingly effective. Used by most repair approaches.
- Usually restricted to single-edit patches. (Combinatorial explosion.)
- Treats the search as a *decision problem*: should I apply this patch?
- Allows us to *substantially* reduce the cost of candidate patch evaluation.

[1] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The strength of random search on automated program repair. In Proceedings of the 36th International Conference on Software Engineering (ICSE 2014). ACM, New York, NY, USA, 254-265. DOI: https://doi.org/10.1145/2568225.2568254
[2] Westley Weimer, Zachary P. Fry, and Stephanie Forrest. 2013. Leveraging program equivalence for adaptive program repair: models and first results. In Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13). IEEE Press, Piscataway, NJ, USA, 356-366. DOI: https://doi.org/10.1109/ASE.2013.6693094

How can we reduce the cost of patch validation?

```
 1: int getYear(int days) {
 2:  int year = 1980;
 3:  while (days > 365) {
 4:   if (isLeapYear(year)) {
 5:    if (days > 366) {
 6:     days -= 366;
 7:     year += 1;
 8:    }
 9:   } else {
10:    days -= 365;
11:    year += 1;
12:   }
13:  }
14:  return year;
15: }
```

```
assert getYear(365) == 1980;
```

```
assert getYear(0) == 1980;
```

```
assert getYear(367) == 1981;
```

```
assert getYear(1826) == 1984;
```

```
assert getYear(1827) == 1984;
```

```
assert getYear(366) == 1980;
```

Random and Exhaustive Search: Optimisations

1. **Avoid redundant tests:** Only execute tests that cover the modified parts of the program.
2. **Short-circuit test suite evaluation:** Stop evaluating the rest of the test suite on the first instance of failure -- the outcome of the test suite evaluation (i.e., PASS or FAIL) won't be changed.
3. **Minimise number of test executions:** Use (online) test prioritisation to minimise the expected number of test executions.

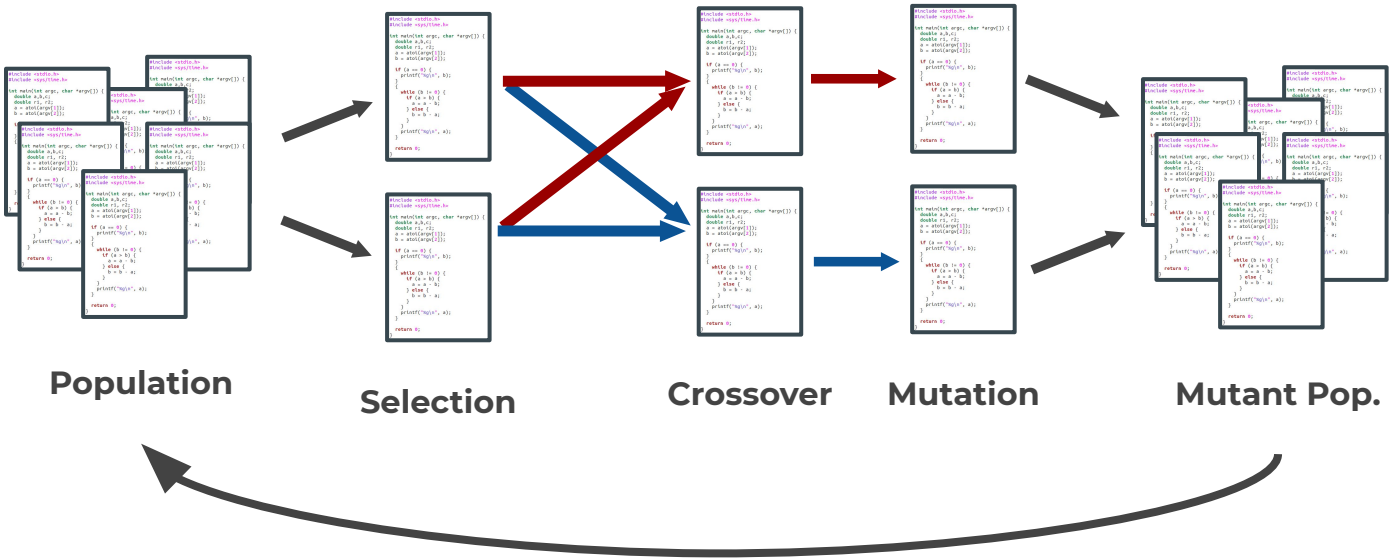
GenProg's Genetic Algorithm [1]

Uses feedback from patch validation to evolve ***multi-edit patches***.

- Evolves a population of individuals that represent candidate patches.
- Each individual is represented as a sequence of edits.
- Each individual has a fitness value that is used to guide the search.
  - Indicates proximity to an acceptable repair.
  - Based on outcomes of test suite.

[1] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. IEEE Transactions on Software Engineering 38, 1 (January 2012), 54-72. DOI: http://dx.doi.org/10.1109/TSE.2011.104

Genetic Algorithm



Fitness Function

Computes a weighted sum of number of positive/negative tests that passed.

$$f = W_{pos} T_{pos} + W_{neg} T_{neg}$$

Rewards fixing the bug $W_{neg} T_{neg}$, and *punishes* regressions $W_{pos} T_{pos}$

Problems:

- In practice, we usually only have a single negative test.
- Lack of gradient to guide the search. Difficult to identify partial solutions.
- Expensive to compute!

Exercise 2: A Cryptic Checksum Conundrum

```
while (next != '\n') {
  scanf("%c", &next);
  sum += next;
}
sum = sum % 64 + 22;
return sum;
```

Takes a single line as input. Should sum the integer codes of the ASCII characters, excluding the newline, modulo 64, plus the code for the space character.

As before, you may only use the following repair operators:

- Delete Statement
- Replace Statement
- Append Statement

Hint: more than one edit is required.

```
while (next != '\n') { ... }
```

```
if (next == '\n') { break; }
```

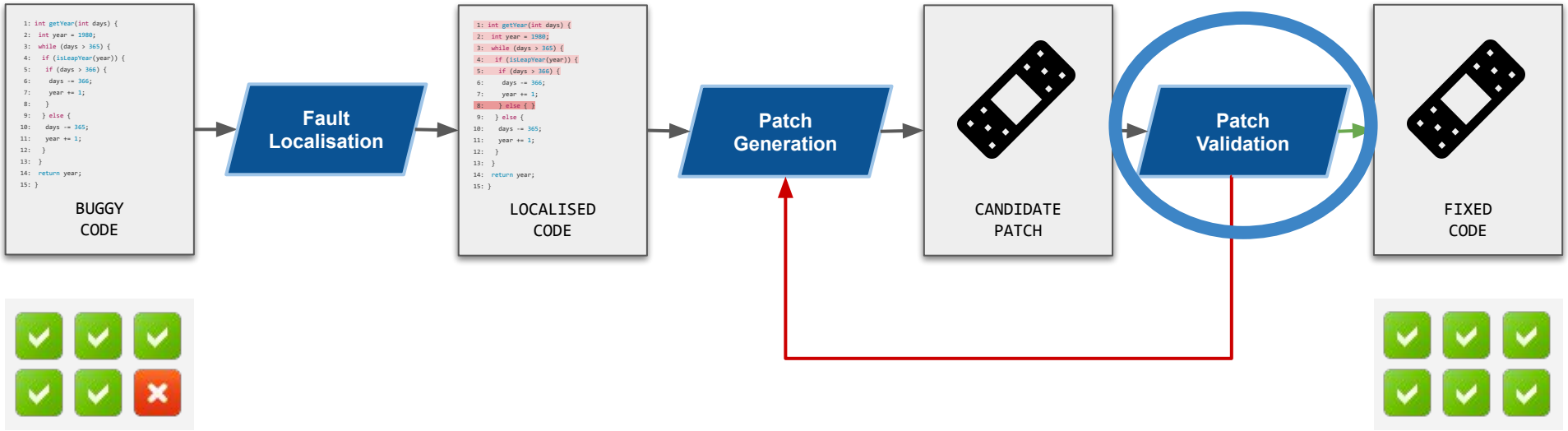
```
scanf("%c", &next);
```

```
sum += next;
```

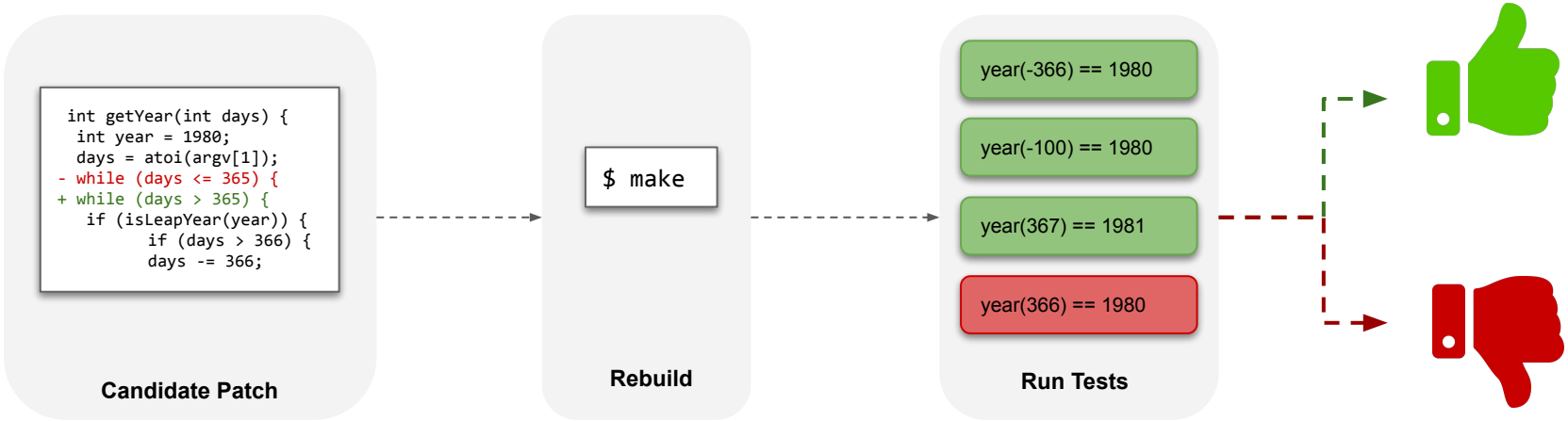
```
sum = sum % 64 + 22;
```

```
return sum;
```

Vinicius Paulo L. Oliveira, Eduardo Faria de Souza, Claire Le Goues and Celso G. Camilo-Junior. Improved representation and genetic operators for linear genetic programming for automated program repair. EMSE, 2018.

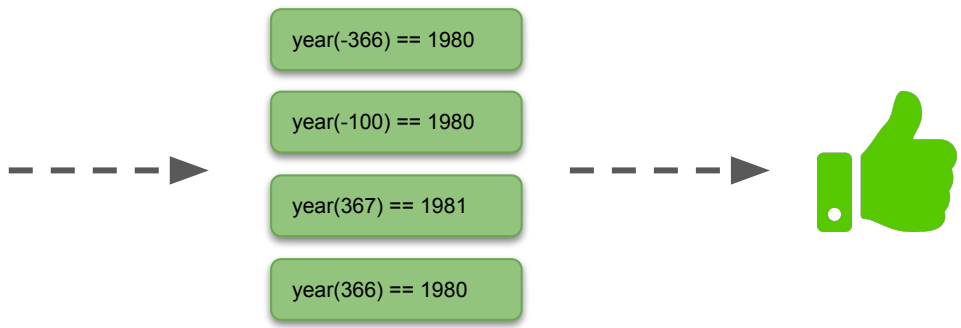

1: int getYear(int days) {
2: int year = 1980;
3: while (days > 365) {
4: if (isLeapYear(year)) {
5: if (days > 366) {
6: days -= 366;
7: year += 1;
8: }
9: } else {
10: days -= 365;
11: year += 1;
12: }
13: }
14: return year;
15: }
BUGGY CODE
Fault Localisation
1: int getYear(int days) {
2: int year = 1980;
3: while (days > 365) {
4: if (isLeapYear(year)) {
5: if (days > 366) {
6: days -= 366;
7: year += 1;
8: } else { }
9: } else {
10: days -= 365;
11: year += 1;
12: }
13: }
14: return year;
15: }
LOCALISED CODE
Patch Generation
CANDIDATE PATCH
Patch Validation
FIXED CODE

Patch Validation

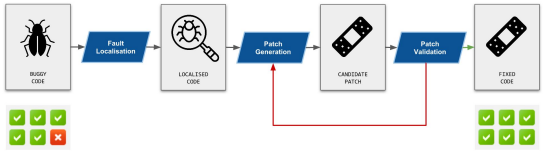


Warning: Don’t trust your test suite

```
 1: int getYear(int days) {
 2:  int year = 1980;
 3:  while (days > 365) {
 4:   if (isLeapYear(year)) {
 5:    if (days > 366) {
 6:     days -= 366;
 7:     year += 1;
 8:    } else { return year; }
 9:   } else {
10:    days -= 365;
11:    year += 1;
12:   }
13:  }
14:  return year;
15: }
```

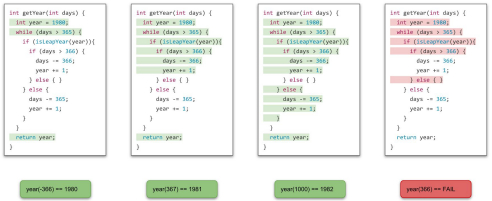


Summary



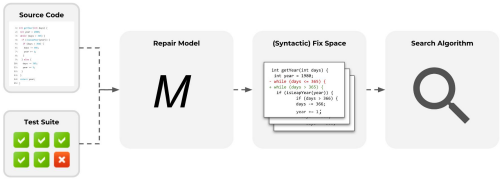
5

SBFL (1/4): Coverage



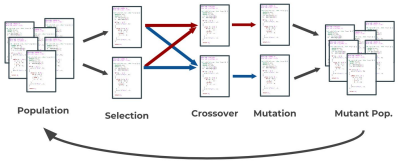
11

Syntax-Based Repair

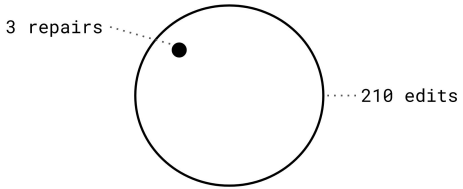


20

Genetic Algorithm



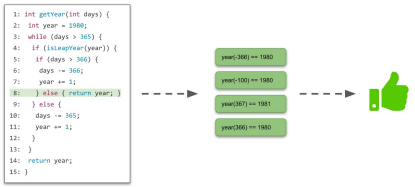
36



Likelihood of picking a repair: **~1.43%**

28

Warning: Don't trust your test suite



41