Leveraging Program Invariants to Promote Population Diversity in Search-Based Automatic Program Repair

Zhen Yu Ding†, Yiwei Lyu‡, Christopher S. Timperley‡, Claire Le Goues‡

† University of Pittsburgh, ‡ Carnegie Mellon University

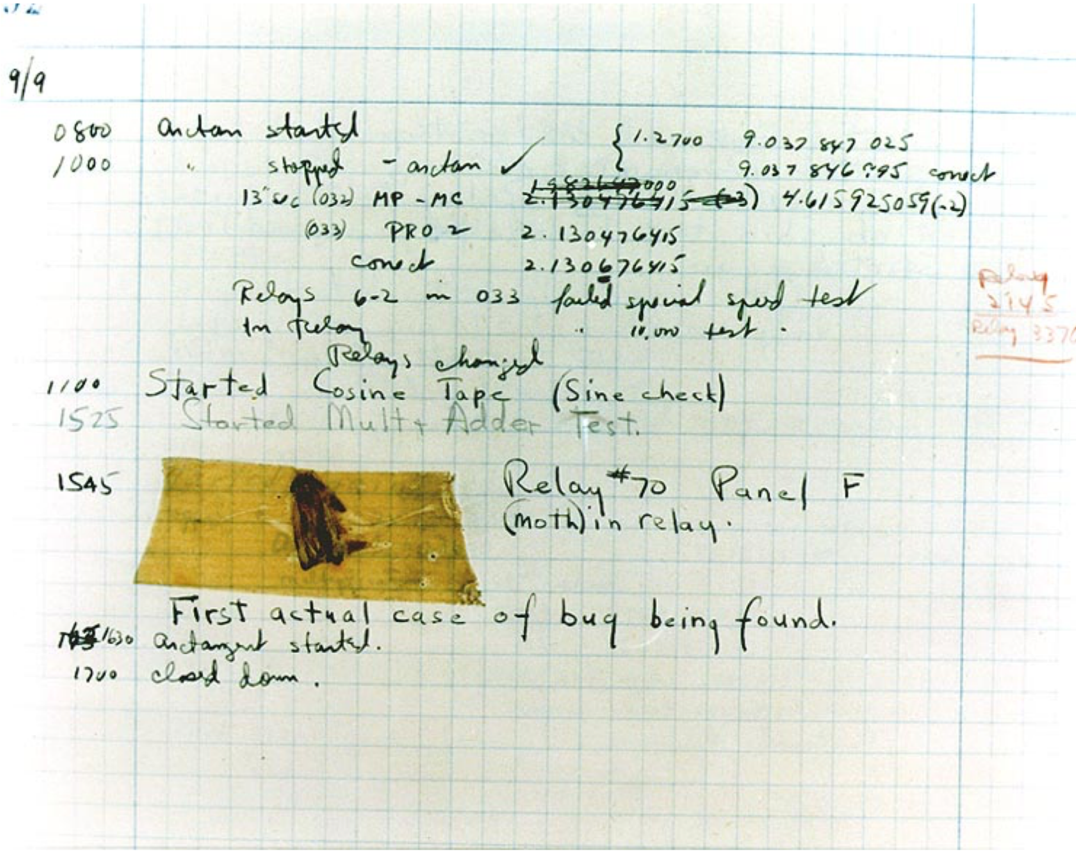
Bugs aren’t great...

In 2017

- 3.7 billion people affected
- Over $1.7 trillion of assets affected

Reduces developer productivity

- Loss of time
- Frustration



Automatic Bug Repair

Angelix

Semantics-based test-driven automated program repair tool for C programs

GenProg

Evolutionary Program Repair

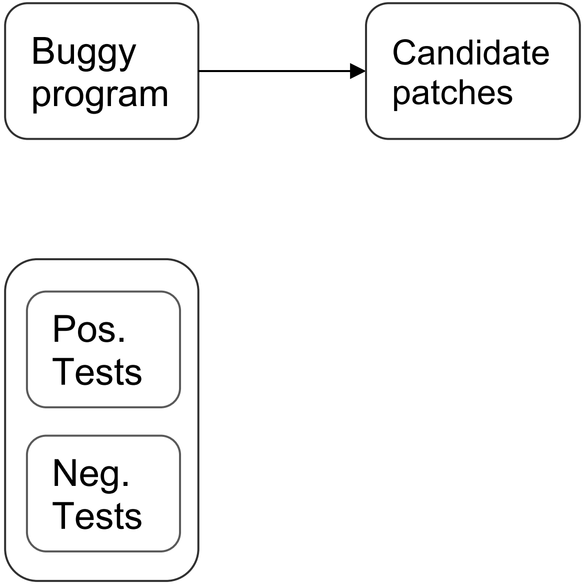
S3: Syntax- and Semantic-Guided Repair Synthesis via Programming by Examples

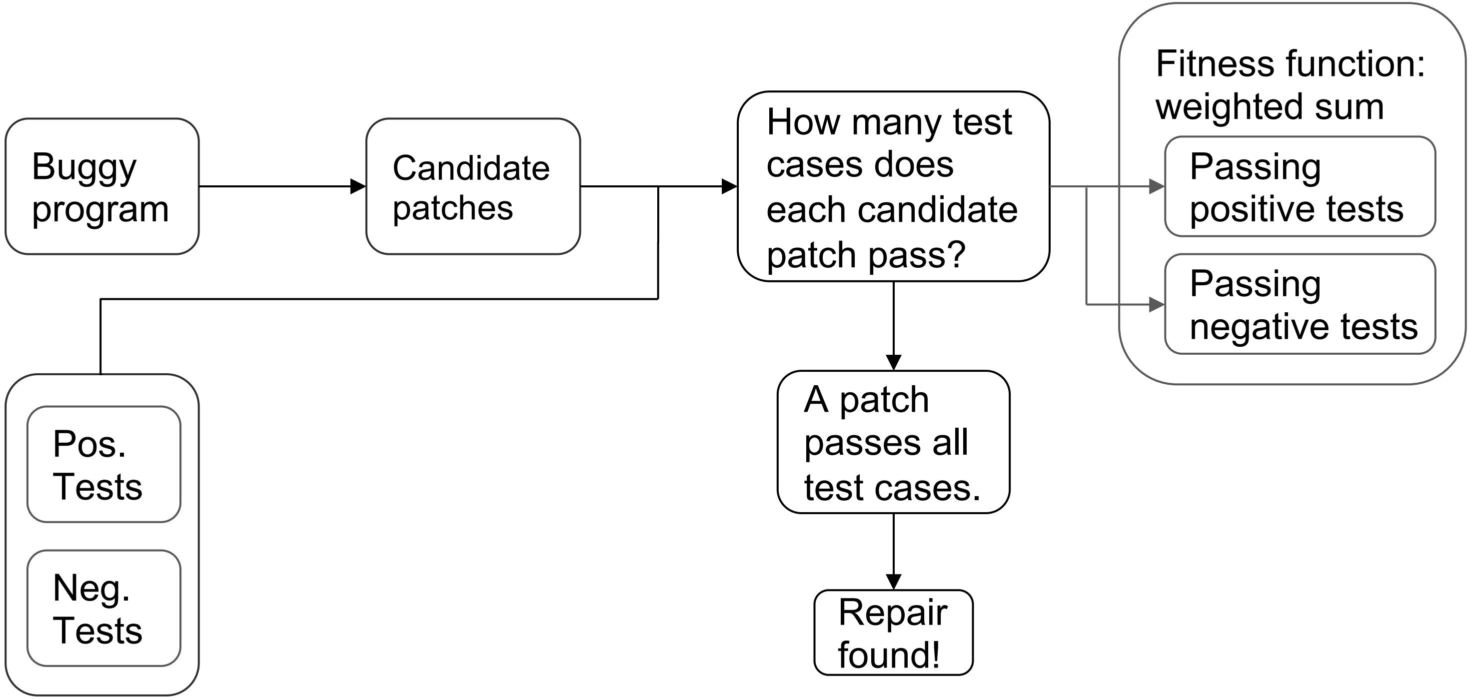
Automatic Bug Repair

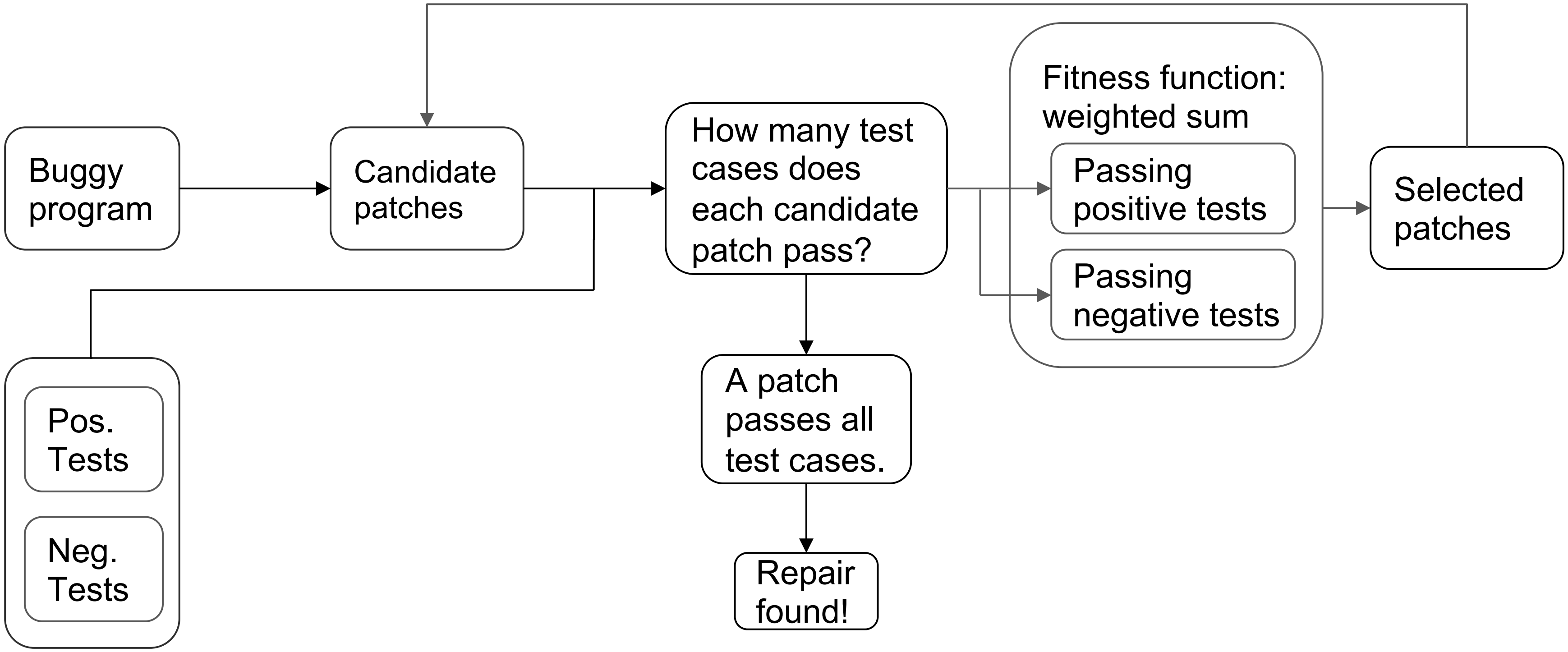
GenProg

Evolutionary Program Repair


Buggy program
Pos. Tests
Neg. Tests


Buggy program
Candidate patches
Pos. Tests
Neg. Tests


Buggy program
Candidate patches
How many test cases does each candidate patch pass?
Fitness function: weighted sum
Passing positive tests
Passing negative tests
Pos. Tests
Neg. Tests
A patch passes all test cases.
Repair found!


Buggy program
Candidate patches
How many test cases does each candidate patch pass?
Fitness function: weighted sum
Passing positive tests
Passing negative tests
Selected patches
Pos. Tests
Neg. Tests
A patch passes all test cases.
Repair found!

```java
public int gcd(int a, int b) {
  int result = 1;
  if (a == 0) {
    b = b - a;
  } else {
    result=a;
    while (b != 0) {
      result = b;
      if (a > b) {
        a = a - b;
      } else {
        b = b - a;
      }
    }
  }
  result=a;
  return result;
}
```

```
public int gcd(int a, int b) {
  int result = 1;
  if (a == 0) {
    b = b - a;
  } else {
    result=a;
    while (b != 0) {
      result = b;
      if (a > b) {
        a = a - b;
      } else {
        b = b - a;
      }
    }
  }
  result=a;
  return result;
}
```

Warning: this is not the normal GCD bug often seen in APR!

```
public int gcd(int a, int b) {
  int result = 1;
  if (a == 0) {
    b = b - a;
  } else {
    result=a;
    while (b != 0) {
      result = b;
      if (a > b) {
        a = a - b;
      } else {
        b = b - a;
      }
    }
  }
  result=a;
  return result;
}
```

```java
public int gcd(int a, int b) {
  int result = 1;
  if (a == 0) {
    b = b - a;
  } else {
    result=a;
    while (b != 0) {
      result = b;
      if (a > b) {
        a = a - b;
      } else {
        b = b - a;
      }
    }
  }
  result=a;
  return result;
}
```

```java
public int gcd(int a, int b) {
  int result = 1;
  if (a == 0) {
    b = b - a;
  } else {
    result=a;
    while (b != 0) {
      result = b;
      if (a > b) {
        a = a - b;
      } else {
        b = b - a;
      }
    }
  }
  result=a;
  return result;
}
```

```
public int gcd(int a, int b) {
  int result = 1;
  if (a == 0) {
    b = b - a;
  } else {
    result=a;
    while (b != 0) {
      result = b;
      if (a > b) {
        a = a - b;
      } else {
        b = b - a;
      }
    }
  }
  result=a;
  return result;
}
```

```java
public int gcd(int a, int b) {
  int result = 1;
  if (a == 0) {
    b = b - a;
  } else {
    result=a;
    while (b != 0) {
      result = b;
      if (a > b) {
        a = a - b;
      } else {
        b = b - a;
      }
    }
  }
  result=a;
  return result;
}
```

Works correctly when `a != 0`

Should return `b` when `a = 0`

This program returns `0` instead

```java
public int gcd(int a, int b) {
  int result = 1;
  if (a == 0) {
    b = b - a;
  } else {
    result=a;
    while (b != 0) {
      result = b;
      if (a > b) {
        a = a - b;
      } else {
        b = b - a;
      }
    }
  }
  result=a;
  return result;
}
```

Works correctly when `a != 0`

Should return `b` when `a = 0`

This program returns `0` instead

Test Cases:

a	b	Expected result	Actual Result	Passed?
5	7	1		
0	2	2		
12	16	4		
3	0	3		
0	10	10		
...	...	...		

```java
public int gcd(int a, int b) {
  int result = 1;
  if (a == 0) {
    b = b - a;
  } else {
    result=a;
    while (b != 0) {
      result = b;
      if (a > b) {
        a = a - b;
      } else {
        b = b - a;
      }
    }
  }
  result=a;
  return result;
}
```

Works correctly when `a != 0`

Should return `b` when `a = 0`

This program returns `0` instead

Test Cases:

a	b	Expected result	Actual Result	Passed?
5	7	1	1	Yes
0	2	2	0	No
12	16	4	4	Yes
3	0	3	3	Yes
0	10	10	0	No
...	...	...	...	...

```
public int gcd(int a, int b) {
  int result = 1;
  if (a == 0) {
    b = b - a;
  } else {
    result=a;
    while (b != 0) {
      result = b;
      if (a > b) {
        a = a - b;
      } else {
        b = b - a;
      }
    }
  }
  result=a;
  return result;
}
```

Problem:

Should return `b` when `a` is `0`

This program returns `0` instead

```
1   public int gcd(int a, int b) {
2     int result = 1;
3     if (a == 0) {
4       b = b - a;  result=b;
5     } else {
6       result=a;
7       while (b != 0) {
8         result = b;
9         if (a > b) {
10          a = a - b;
11        } else {
12          b = b - a;
13        }
14      }
15    }
16    result=a;
17    return result;
18  }
```

Problem:

Should return `b` when `a` is `0`

This program returns `0` instead

Simplest fix is 2 steps:

(1) Delete line 16

(2) Replace line 4 with line 8

```java
1  public int gcd(int a, int b) {
2    int result = 1;
3    if (a == 0) {
4      b = b - a;
5    } else {
6      result=a;
7      while (b != 0) {
8        result = b;
9        if (a > b) {
10         a = a - b;
11       } else {
12         b = b - a;
13       }
14     }
15   }
16   result=a;
17   return result;
18 }
```

Simplest fix is 2 steps:

(1) Delete line 16

~~(2) Replace line 4 with line 8~~

If we only perform step 1 (partial repair):

```
1  public int gcd(int a, int b) {
2    int result = 1;
3    if (a == 0) {
4      b = b - a;
5    } else {
6      result=a;
7      while (b != 0) {
8        result = b;
9        if (a > b) {
10         a = a - b;
11       } else {
12         b = b - a;
13       }
14     }
15   }
16   result=a;
17   return result;
18 }
```

Simplest fix is 2 steps:

(1) Delete line 16

~~(2) Replace line 4 with line 8~~

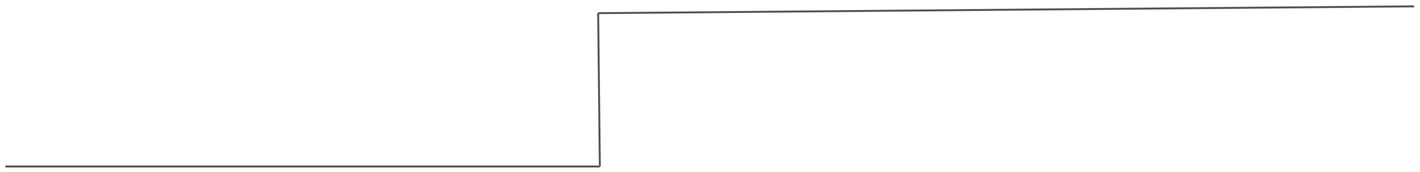
If we only perform step 1 (partial repair):

- Still fails when `a=0`, passes otherwise
- Cannot be differentiated just from test results.

Patch indistinguishability

Test cases often fail to distinguish between different candidate patches.

Plateau-like fitness landscape.



S. Forrest, W. Weimer, T. Nguyen, and C. Le Goues, “A genetic programming approach to automated software repair,” in Genetic and Evolutionary Computation Conference (GECCO), 2009, pp. 947–954.

E. Fast, C. Le Goues, S. Forrest, and W. Weimer, “Designing better fitness functions for automated program repair,” in Genetic and Evolutionary Computation Conference, ser. GECCO ’10, 2010, pp. 965–972.

E. F. de Souza, C. Le Goues, and C. G. Camilo-Junior, “A novel fitness function for automated program repair based on source code checkpoints,” in Genetic and Evolutionary Computation Conference, ser. GECCO ’18, 2018.

Goal: distinguish patches better

Goal: distinguish patches better

Infer invariants to semantically describe candidate patches.

Find semantically unique/diverse candidate patches.

An intuition on why.

```
public int gcd(int a, int b) {
  int result = 1;
  if (a == 0) {
    b = b - a;
  } else {
    result=a;
    while (b != 0) {
      result = b;
      if (a > b) {
        a = a - b;
      } else {
        b = b - a;
      }
    }
  }
  result=a;
  return result;
}
```

An intuition on why.

```java
public int gcd(int a, int b) {
  int result = 1;
  if (a == 0) {
    b = b - a;
  } else {
    result=a;
    while (b != 0) {
      result = b;
      if (a > b) {
        a = a - b;
      } else {
        b = b - a;
      }
    }
  }
  result=a;
  return result;
}
```

Invariants when running positive tests (gcd(5,7), gcd(12,16), gcd(3,0), etc):

- `a>=0`
- `b>=0`
- `result>=0`
- `a%result==0`
- `b%result==0`
- …

An intuition on why.

```
1  public int gcd(int a, int b) {
2    int result = 1;
3    if (a == 0) {
4      b = b - a; result=b;
5    } else {
6      result=a;
7      while (b != 0) {
8        result = b;
9        if (a > b) {
10         a = a - b;
11       } else {
12         b = b - a;
13       }
14     }
15   }
16   result=a;
17   return result;
18 }
```

One simple fix:

(1) Delete line 16

(2) Replace line 4 with line 8

An intuition on why.

```
1  public int gcd(int a, int b) {
2    int result = 1;
3    if (a == 0) {
4      b = b - a;
5    } else {
6      result=a;
7      while (b != 0) {
8        result = b;
9        if (a > b) {
10         a = a - b;
11       } else {
12         b = b - a;
13       }
14     }
15   }
16   result=a;
17   return result;
18 }
```

One simple fix:

(1) Delete line 16
~~(2) Replace line 4 with line 8~~

If we only perform step 1 (partial repair):

- Still fails when `a=0`, passes otherwise
- Cannot be differentiated just from test results.

An intuition on why.

```
public int gcd(int a, int b) {
  int result = 1;
  if (a == 0) {
    b = b - a;
  } else {
    result=a;
    while (b != 0) {
      result = b;
      if (a > b) {
        a = a - b;
      } else {
        b = b - a;
      }
    }
  }
  result=a;
  return result;
}
```

Invariant `a%result==0`:

- True when `a != 0`
- False when `a=0` (result is `0`)

An intuition on why.

```
public int gcd(int a, int b) {
  int result = 1;
  if (a == 0) {
    b = b - a;
  } else {
    result=a;
    while (b != 0) {
      result = b;
      if (a > b) {
        a = a - b;
      } else {
        b = b - a;
      }
    }
  }
  result=a;
  return result;
}
```

Invariant `a%result==0`:

- True when `a != 0`
- False when `a=0` (result is `0`)
- True when `a=0` in partial repair

An intuition on why.

```
public int gcd(int a, int b) {
  int result = 1;
  if (a == 0) {
    b = b - a;
  } else {
    result=a;
    while (b != 0) {
      result = b;
      if (a > b) {
        a = a - b;
      } else {
        b = b - a;
      }
    }
  }
  result=a;
  return result;
}
```

Invariant `a%result==0`:

- True when `a != 0`
- False when `a=0` (result is `0`)
- True when `a=0` in partial repair

Partial repair results in invariant behavior change!

Daikon – an invariant detection tool

A mature dynamic invariant detection technique

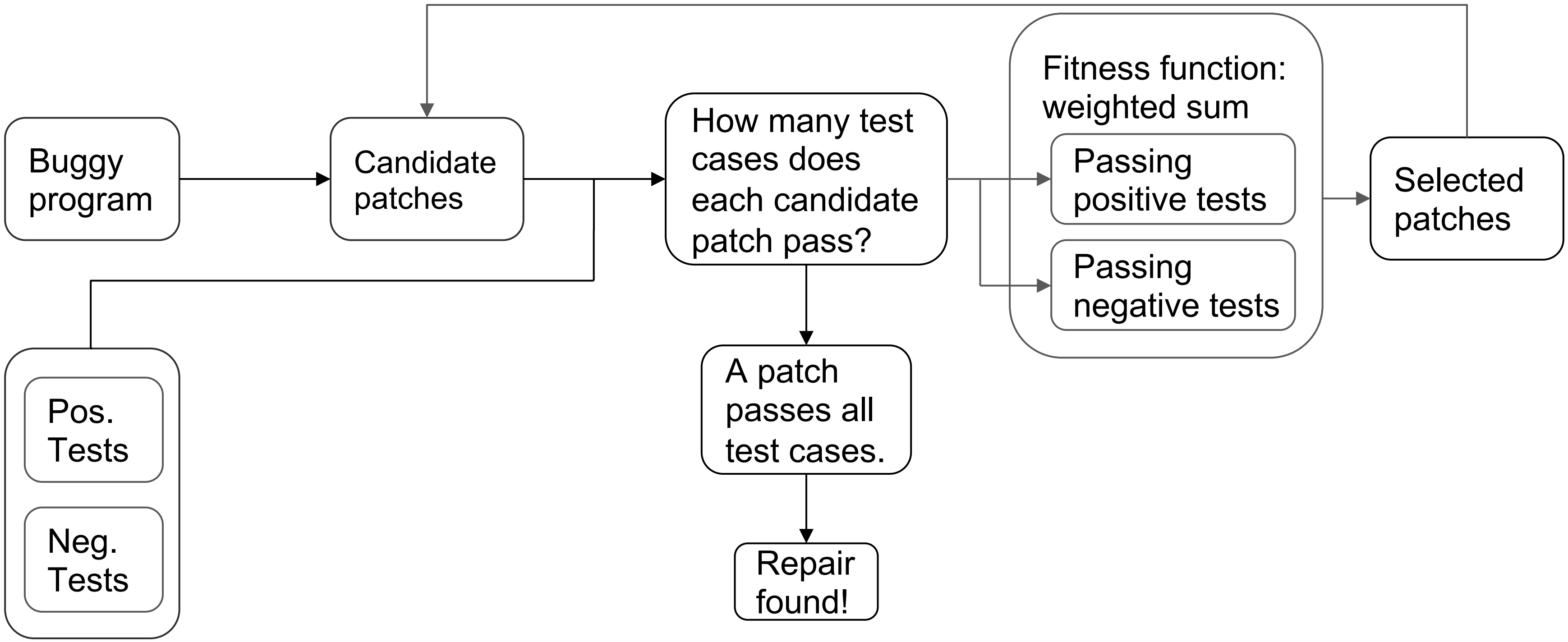
- Runs the program and record traces of intermediate variable values
- Analyze the traces to learn invariants

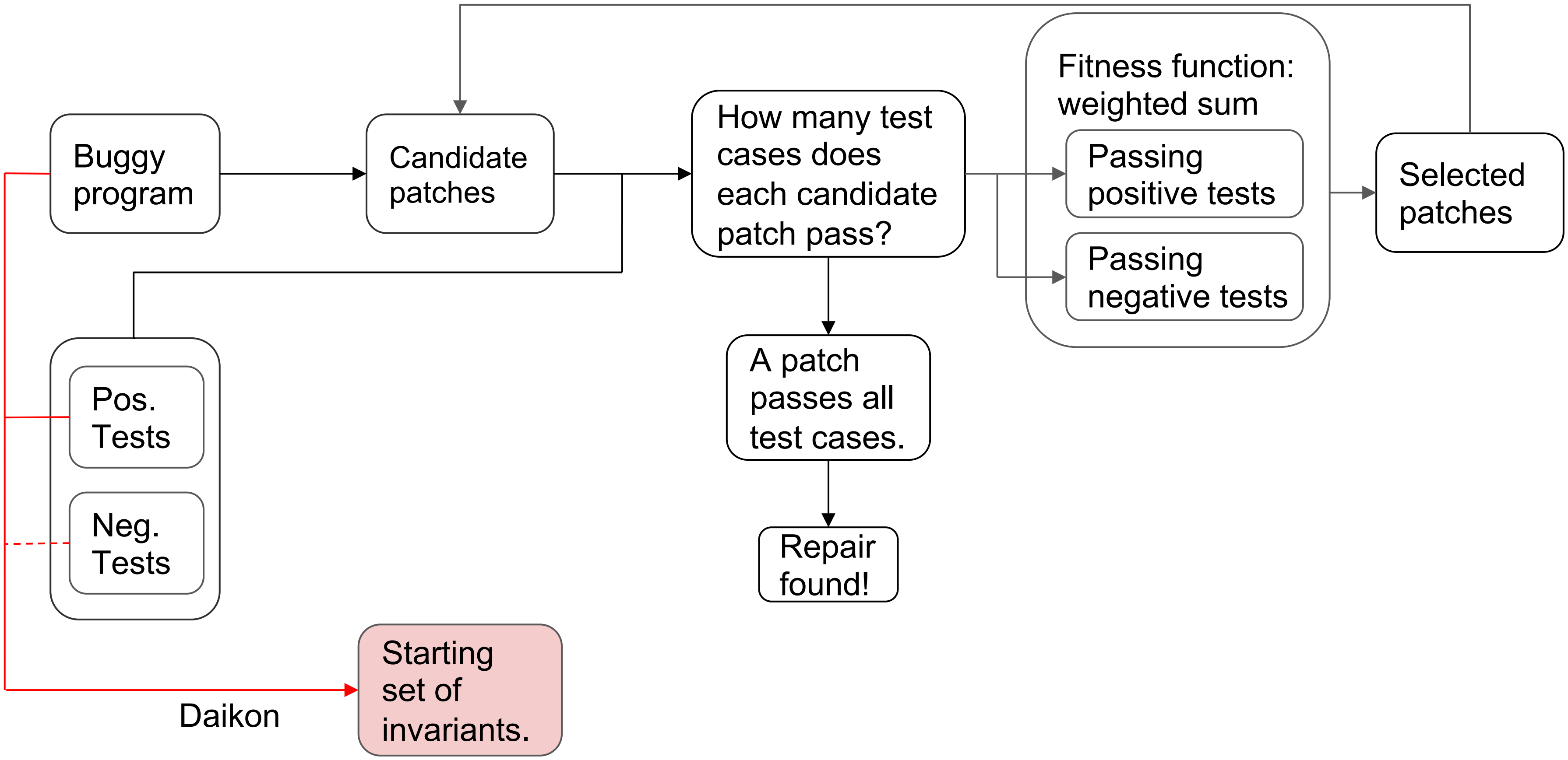
```java
public int gcd(int a, int b) {
  int result = 1;
  if (a == 0) {
    b = b - a;
  } else {
    result=a;
    while (b != 0) {
      result = b;
      if (a > b) {
        a = a - b;
      } else {
        b = b - a;
      }
    }
  }
  result=a;
  return result;
}
```

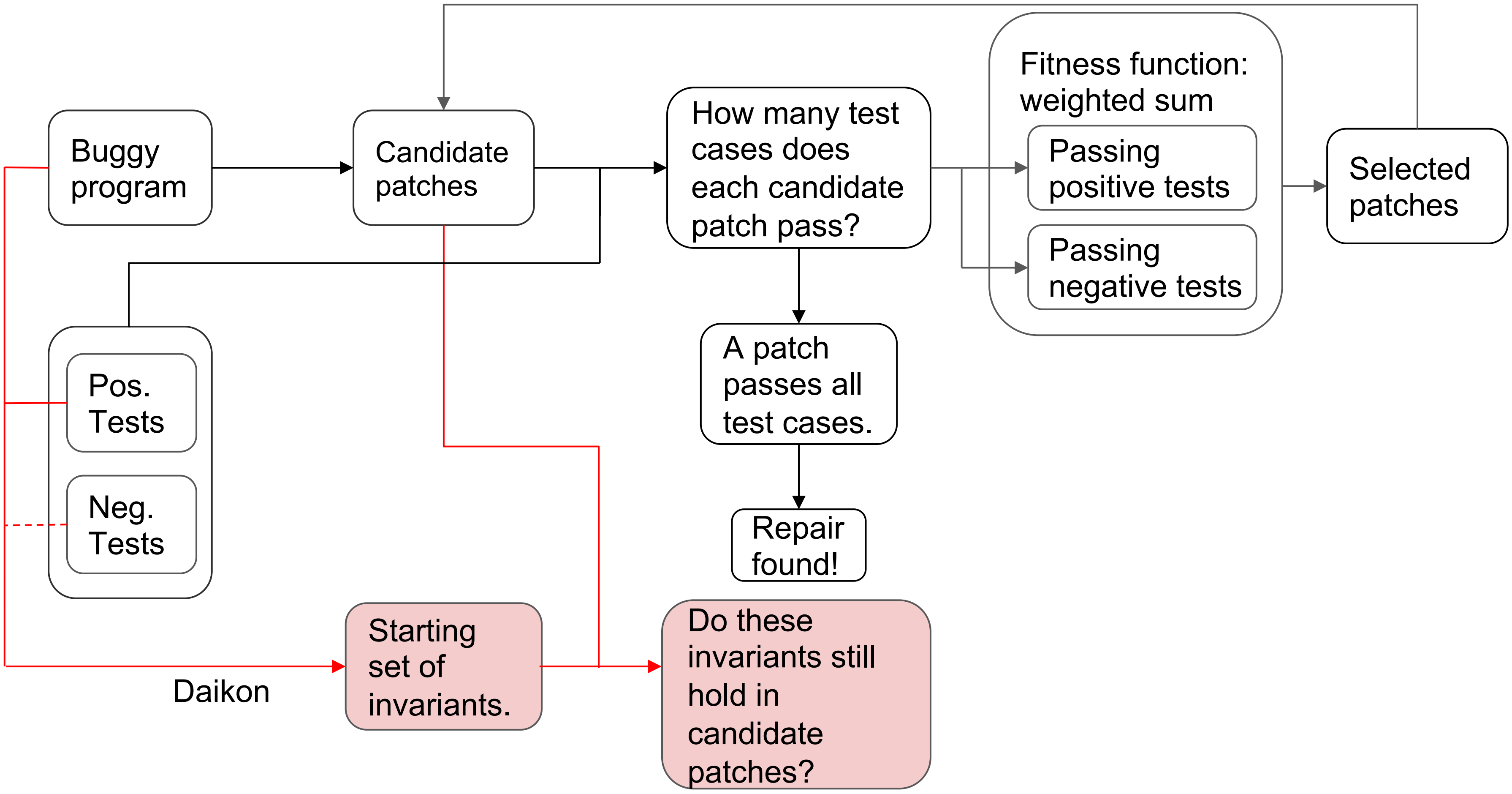
Invariants when running positive tests (gcd(5,7), gcd(12,16), gcd(3,0), etc):

- `a>=0`
- `b>=0`
- `result>=0`
- `a%result==0`
- `b%result==0`
- ...

All were detected by Daikon


Buggy program
Candidate patches
How many test cases does each candidate patch pass?
Fitness function: weighted sum
Passing positive tests
Passing negative tests
Selected patches
Pos. Tests
Neg. Tests
A patch passes all test cases.
Repair found!


Buggy program
Candidate patches
How many test cases does each candidate patch pass?
Fitness function: weighted sum
Passing positive tests
Passing negative tests
Selected patches
Pos. Tests
Neg. Tests
A patch passes all test cases.
Repair found!
Daikon
Starting set of invariants.



Buggy program
Candidate patches
How many test cases does each candidate patch pass?
Fitness function: weighted sum
Passing positive tests
Passing negative tests
Selected patches
Pos. Tests
Neg. Tests
A patch passes all test cases.
Repair found!
Daikon
Starting set of invariants.
Do these invariants still hold in candidate patches?

Starting set of invariants		
a%result==0		
b%result==0		
result>=0		

Starting set of invariants	Candidate patch 0	
`a%result==0`		
`b%result==0`		
`result>=0`		

Starting set of invariants	Candidate patch 0	Tested against
`a%result==0`		Pos. tests
`b%result==0`		
`result>=0`		

= Invariant never violated during program execution.

Starting set of invariants	Candidate patch 0	Tested against
`a%result==0`		Pos. tests
	✗	Neg. tests
`b%result==0`		
`result>=0`		

= Invariant never violated during program execution.

✗ = Invariant violated at least once.

Starting set of invariants	Candidate patch 0	Tested against
`a%result==0`		Pos. tests
	✗	Neg. tests
`b%result==0`		Pos. tests
`result>=0`		

= Invariant never violated during program execution.

✗ = Invariant violated at least once.

Starting set of invariants	Candidate patch 0	Tested against
`a%result==0`		Pos. tests
	✗	Neg. tests
`b%result==0`		Pos. tests
	?	Neg. tests
`result>=0`		

= Invariant never violated during program execution.

✗ = Invariant violated at least once.

? = Invariant not testable.

Starting set of invariants	Candidate patch 0	Tested against
`a%result==0`		Pos. tests
	✗	Neg. tests
`b%result==0`		Pos. tests
	?	Neg. tests
`result>=0`		Pos. tests
	✗	Neg. tests

= Invariant never violated during program execution.

✗ = Invariant violated at least once.

? = Invariant not testable.

Starting set of invariants	Candidate patch 0	Candidate patch 1
`a%result==0`		
	✗	
`b%result==0`		
	?	✗
`result>=0`		
	✗	✗

= Invariant never violated during program execution.

✗ = Invariant violated at least once.

? = Invariant not testable.

Starting set of invariants	Candidate patch 0	Candidate patch 1
`a%result==0`		
	✗	
`b%result==0`		
	?	✗
`result>=0`		
	✗	✗

Invariant profile:

Describes the semantics of a program based on a set of predicates.

Starting set of invariants	Candidate patch 0	Candidate patch 1
`a%result==0`		
	✗	
`b%result==0`		
	?	✗
`result>=0`		
	✗	✗

Invariant profile:

Describes the semantics of a program based on a set of predicates.

We use string comparisons to compare program semantics.

- We use Hamming distances.

Starting set of invariants	Candidate patch 0	Candidate patch 1
`a%result==0`		
	✗	
`b%result==0`		
	?	✗
`result>=0`		
	✗	✗

$\Delta(p0, p1) = 2$

Invariant profile:

Describes the semantics of a program based on a set of predicates.

We use string comparisons to compare program semantics.

- We use Hamming distances.

Invariant profile:

Describes the semantics of a program based on a set of predicates.

We can use string comparisons to compare program semantics.

- We use Hamming distances.

We can calculate semantic diversity.

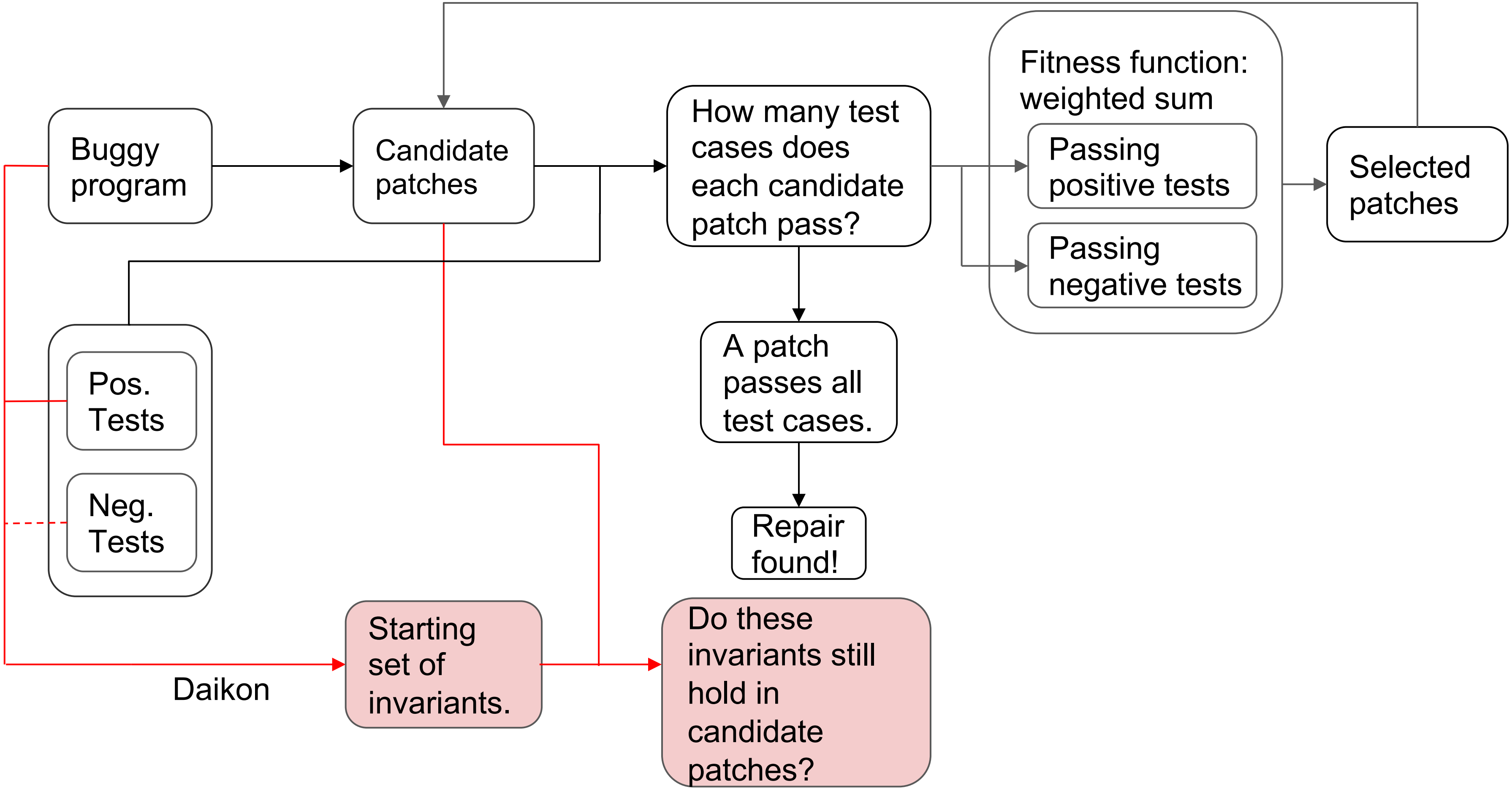
- Sum the Hamming distances.

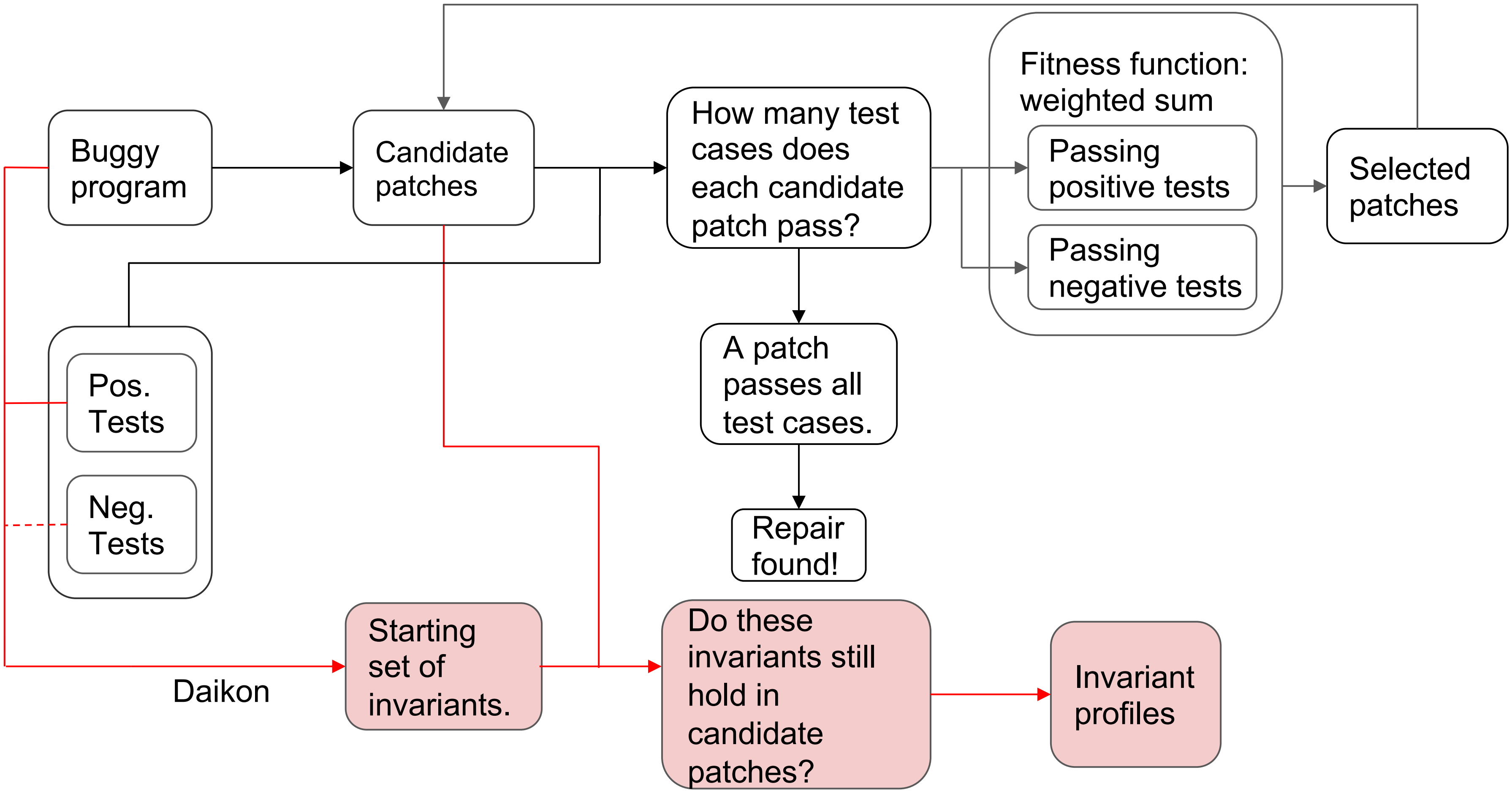
Starting set of invariants	Candidate patch 0	Candidate patch 1	Candidate patch 2
`a%result==0`			✗
	✗		✗
`b%result==0`			
	?	✗	?
`result>=0`			
	✗	✗	✗

Δ(p0, p1) = 2 Δ(p1, p2) = 3

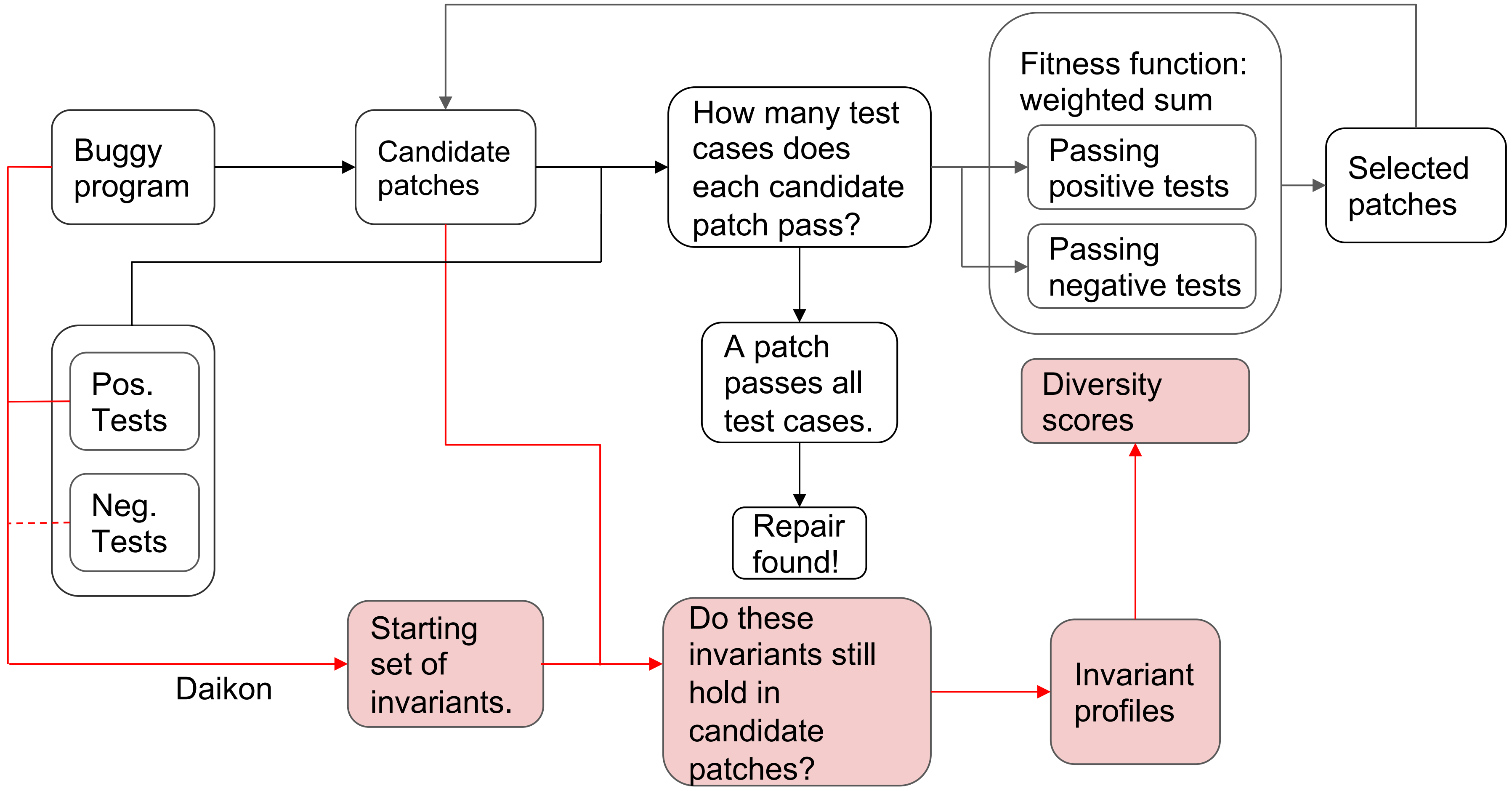
Δ(p0, p2) = 1

diversity(p0) = Δ(p0, p1) + Δ(p0, p2) = 2 + 1 = 3
***diversity*(p1) = Δ(p1, p0) + Δ(p1, p2) = 2 + 3 = 5**
diversity(p2) = Δ(p2, p0) + Δ(p2, p1) = 1 + 3 = 4

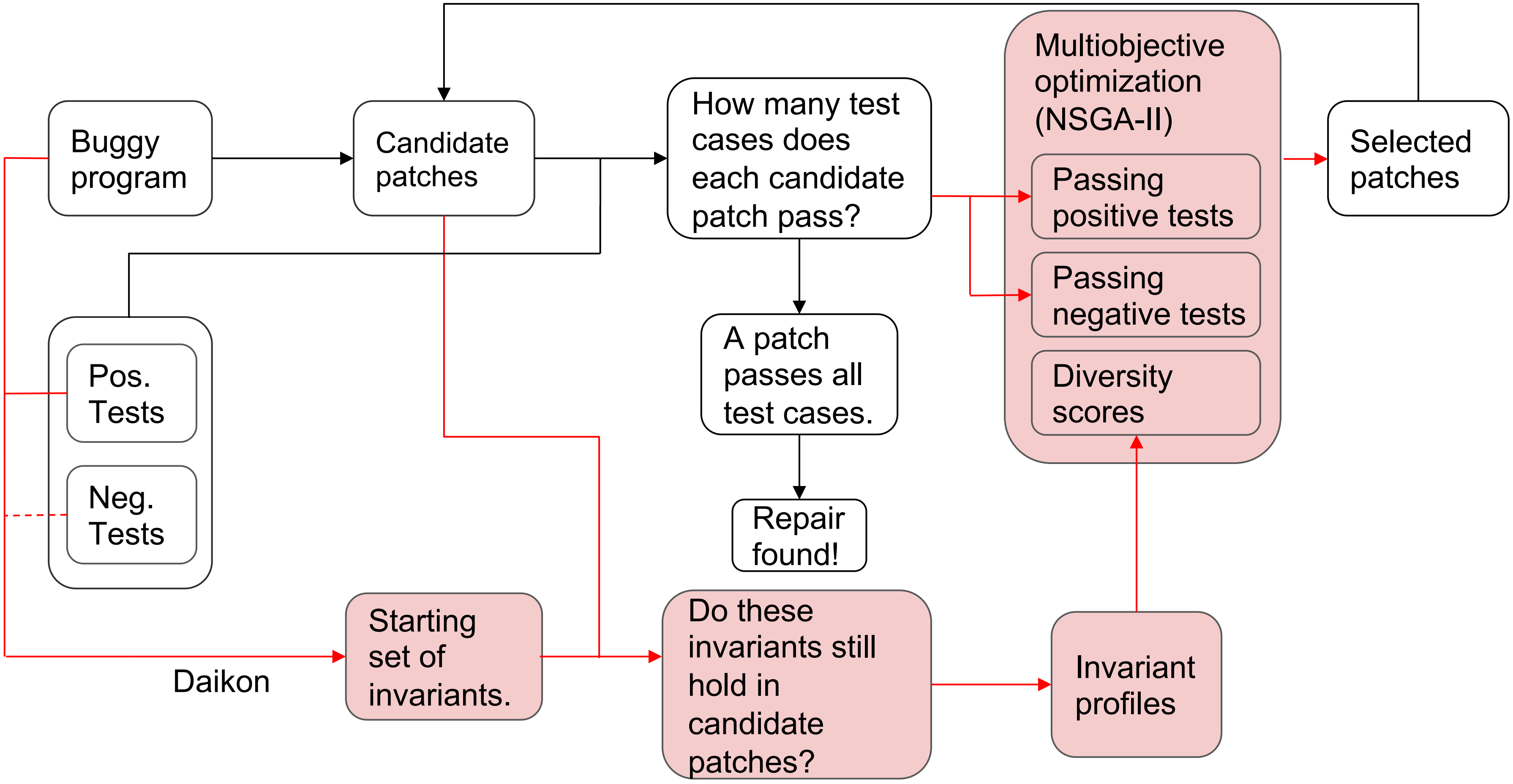

Buggy program
Candidate patches
How many test cases does each candidate patch pass?
Fitness function: weighted sum
Passing positive tests
Passing negative tests
Selected patches
Pos. Tests
Neg. Tests
A patch passes all test cases.
Repair found!
Daikon
Starting set of invariants.
Do these invariants still hold in candidate patches?



Buggy program
Candidate patches
How many test cases does each candidate patch pass?
Fitness function: weighted sum
Passing positive tests
Passing negative tests
Selected patches
Pos. Tests
Neg. Tests
A patch passes all test cases.
Repair found!
Daikon
Starting set of invariants.
Do these invariants still hold in candidate patches?
Invariant profiles



Buggy program
Candidate patches
How many test cases does each candidate patch pass?
Fitness function: weighted sum
Passing positive tests
Passing negative tests
Selected patches
Pos. Tests
Neg. Tests
A patch passes all test cases.
Repair found!
Diversity scores
Daikon
Starting set of invariants.
Do these invariants still hold in candidate patches?
Invariant profiles



Buggy program
Candidate patches
How many test cases does each candidate patch pass?
Multiobjective optimization (NSGA-II)
Passing positive tests
Passing negative tests
Diversity scores
Selected patches
Pos. Tests
Neg. Tests
A patch passes all test cases.
Repair found!
Daikon
Starting set of invariants.
Do these invariants still hold in candidate patches?
Invariant profiles

Evaluation

- IntroClass is a set of small, buggy C programs collected from introductory programming courses.
- IntroClassJava is a subset of IntroClass automatically transformed from C to Java.
- Randomly sampled 59 out of 297 bugs in IntroClassJava for our experiment
- Run each selected bug 10 times with different randomization seeds.

checksum	digits	grade	median	smallest	syllables	**Total**
2/11	14/75	19/89	9/57	13/52	2/13	**59/297**

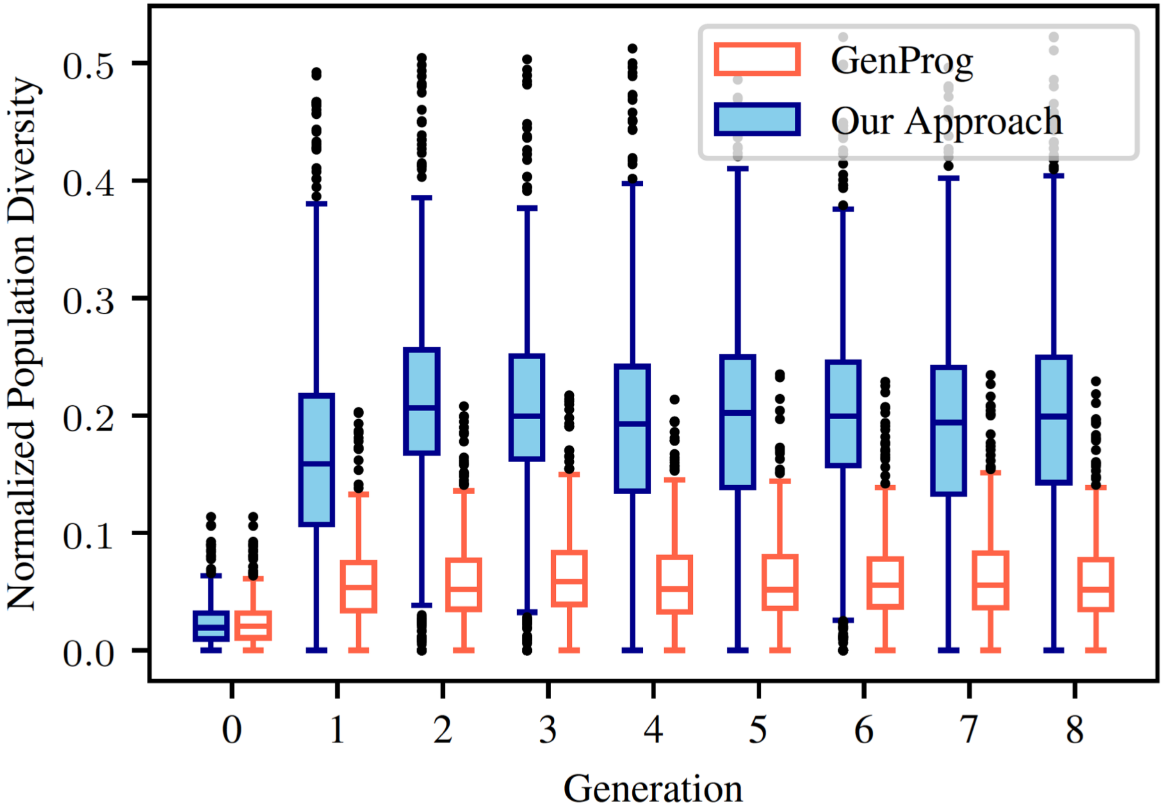
Results

No evidence of improvement in repair performance.

Successfully shown that our approach:

- Promotes semantic diversity
- Improves fitness granularity (therefore reduced plateauing)

GenProg implicitly selects for semantic diversity.



Scalability

IntroClassJava is small (<30 LoC)
Defects4J is large, real-world Java bugs

	Lines of Code	Number of Unit Tests
Apache Commons Math	~85K	3602
Apache Commons Lang	~20K	2245

Scalability

IntroClassJava is small (<30 LoC)
Defects4J is large, real-world Java bugs

	Lines of Code	Number of Unit Tests
Apache Commons Math	~85K	3602
Apache Commons Lang	~20K	2245

- Infeasible to collect invariants by running all thousands of positive tests
- Instead, we only collect invariants by running positive tests co-located in the same test class as the failing test cases.

Scalability

Bug	GenProg Runtime (mins)	Our Approach's Runtime (mins)	Difference
lang11	59.77	64.37	1.08 X
lang29	29.72	37.05	1.25 X
lang36	34.80	41.08	1.18 X
lang8	97.50	103.98	1.07 X
lang9	55.07	70.87	1.29 X
math30	89.27	90.55	1.01 X
math44	98.43	176.88	1.80 X
math46	67.05	720.48	10.75 X
math79	100.55	119.63	1.19 X
math86	62.52	71.45	1.14 X
Median	64.78	81.00	1.19 X
Mean	69.47	149.63	2.18 X

Overheads: invariant learning and checking

Our approach is as scalable as GenProg!

Conclusion

Conclusion

Test cases often can’t distinguish between different patches.

Conclusion

Test cases often can’t distinguish between different patches.

We use inferred invariants to get more semantic information.

Conclusion

Test cases often can't distinguish between different patches.

We use inferred invariants to get more semantic information.

We encourage exploration of semantically diverse patches.

Conclusion

Test cases often can't distinguish between different patches.

We use inferred invariants to get more semantic information.

We encourage exploration of semantically diverse patches.

Invariants can effectively promote diversity & semantic exploration.

Conclusion

Test cases often can’t distinguish between different patches.

We use inferred invariants to get more semantic information.

We encourage exploration of semantically diverse patches.

Invariants can effectively promote diversity & semantic exploration.

No conclusive results on improvements to repair success and efficiency.