

Automated Documentation Inference to Explain Failed Tests

Sai Zhang

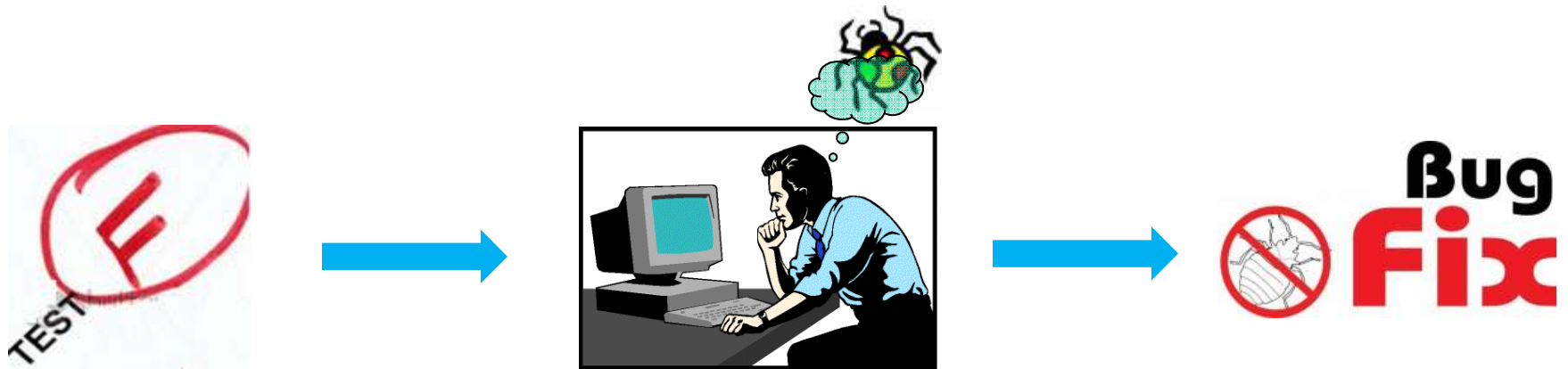
University of Washington

Joint work with: Cheng Zhang, Michael D. Ernst

A failed test reveals a potential bug

Before bug-fixing, programmers must:

- find code relevant to the failure
- understand why the test fails



*Programmers often need to **guess** about relevant parts in the **test** and **tested code***

- Long test code
- Multiple class interactions
- Poor documentation

A failed test

```
public void test1() {  
    int i = 1;  
    ArrayList lst = new ArrayList(i);  
    Object o = new Object();  
    boolean b = lst.add(o);  
    TreeSet ts = new TreeSet(lst);  
    Set set = Collections.synchronizedSet(ts);  
    assertTrue(set.equals(set)); X  
}
```



Which parts of the test are most relevant to the failure?

(The test is minimized, and does not dump a useful stack trace.)

FailureDoc: inferring explanatory documentation

- FailureDoc infers **debugging clues**:
 - Indicates changes to the test that will make it pass
 - Helps programmers understand why the test fails



- FailureDoc provides a **high-level** description of the failure from the perspective of the test
 - Automated fault localization tools pinpoint the buggy statements without explaining why

Documenting the failed test

(The **red** part is generated by **FailureDoc**)

```
public void test1() {
    int i = 1;
    ArrayList lst = new ArrayList(i);
    //Test passes if o implements Comparable
    Object o = new Object();
    //Test passes if o is not added to lst
    boolean b = lst.add(o);
    TreeSet ts = new TreeSet(lst);
    Set set = Collections.synchronizedSet(ts);
    assertTrue(set.equals(set));
}
```

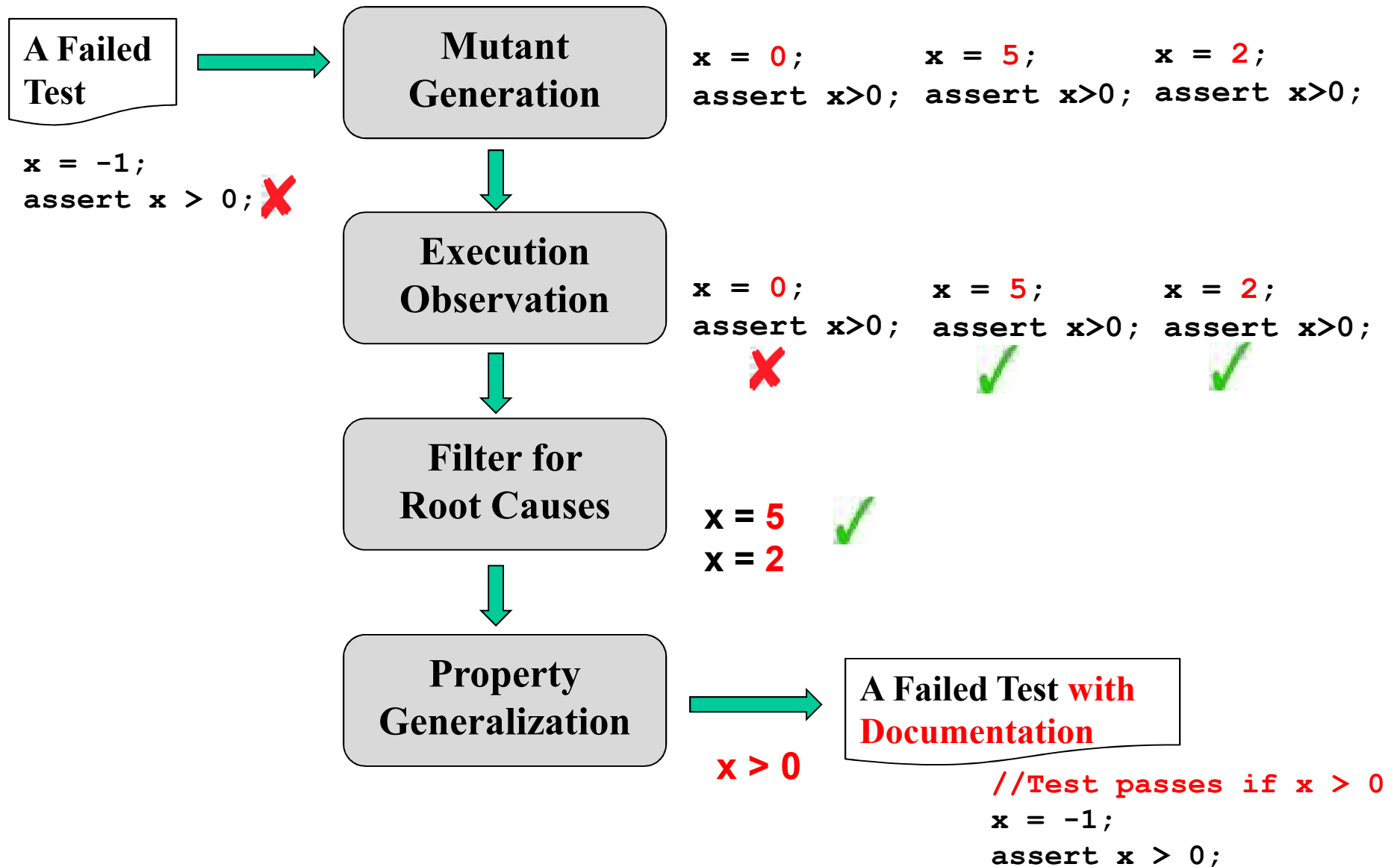
The documentation indicates:

- The `add` method should not accept a **non-Comparable** object, but it does.
- It is a **real** bug.

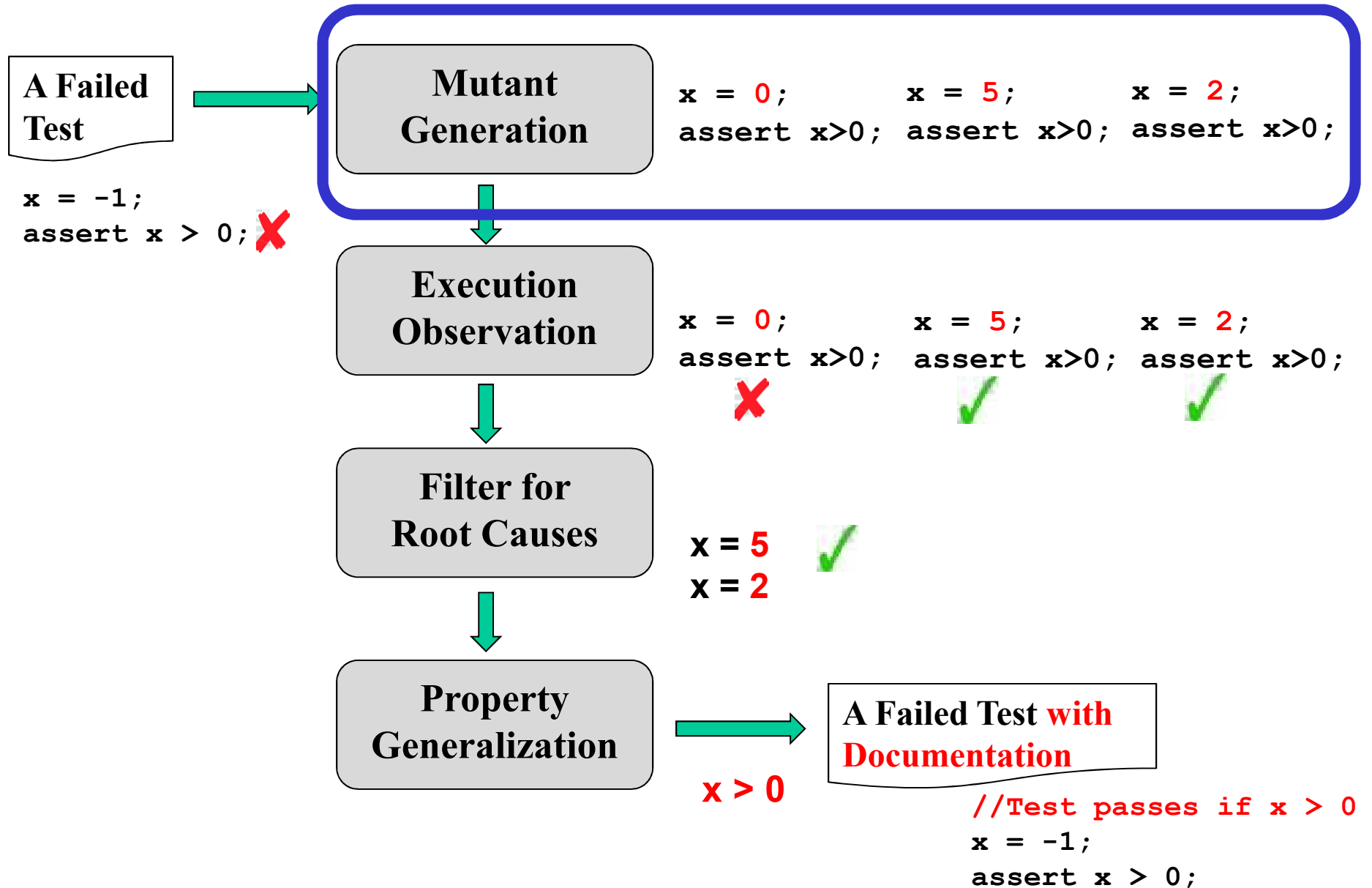
Outline

- Overview
- • The **FailureDoc** technique
- Implementation & Evaluation
- Related work
- Conclusion

The architecture of FailureDoc



The architecture of FailureDoc



Mutant generation via value replacement

- Mutate the failed test by repeatedly replacing an existing input value with **an alternative one**
 - Generate a set of *slightly different* tests

Original test

```
...  
Object o = new Object();  
boolean b = lst.add(o);  
...
```



Mutated test

```
...  
Object o = new Integer(1);  
boolean b = lst.add(o);  
...
```

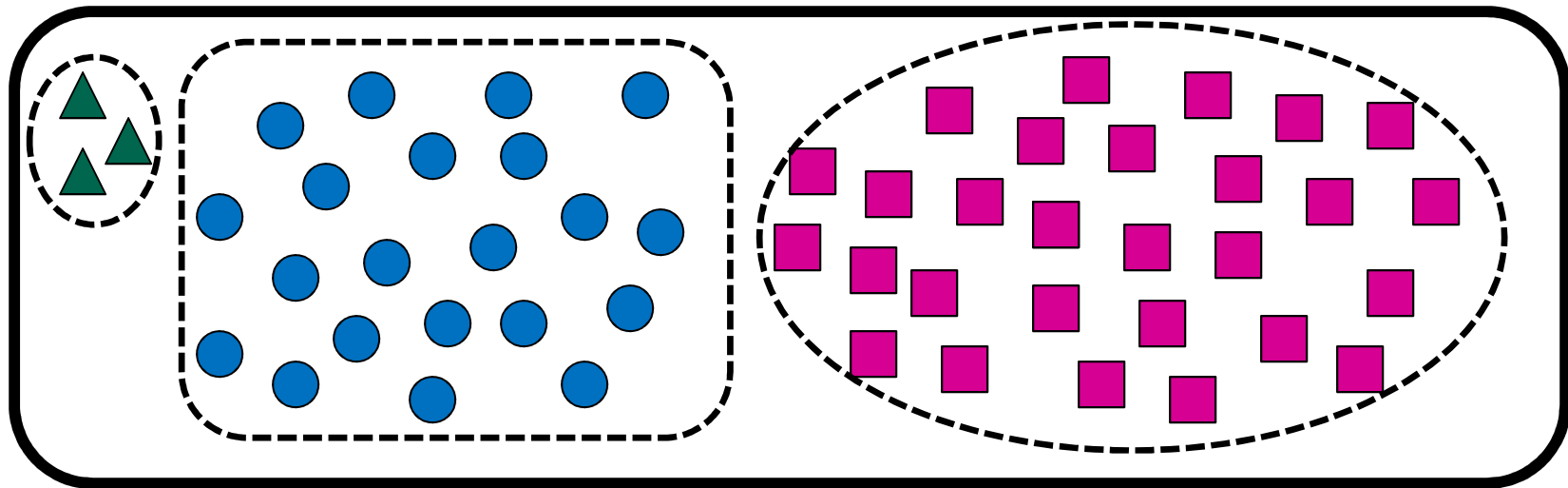
```
...  
TreeSet t = new TreeSet(1);  
Set s = synchronizedSet(t);  
...
```



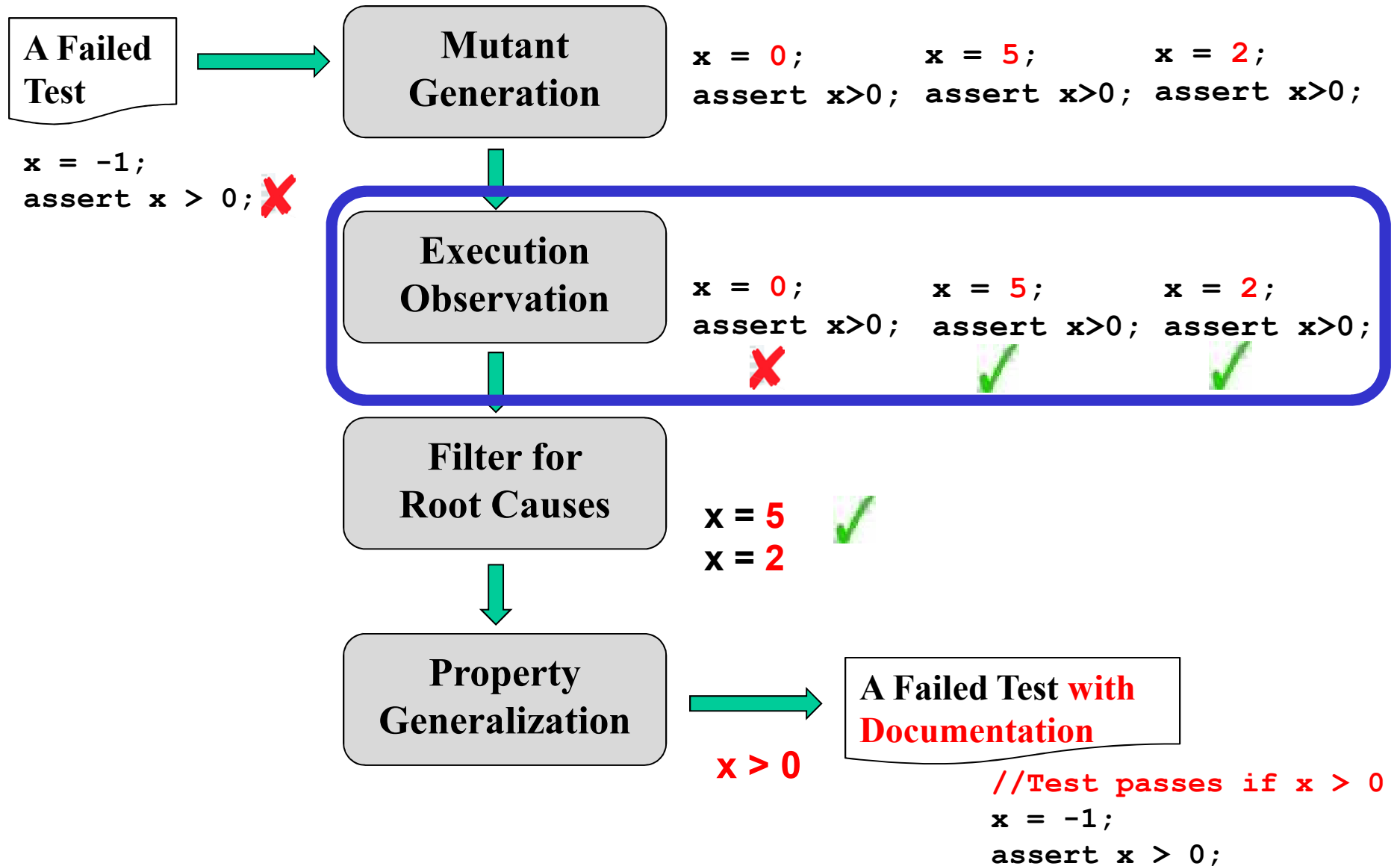
```
...  
TreeSet t = new TreeSet();  
t.add(10);  
Set s = synchronizedSet(t);  
...
```

Value selection in replacement

- **Exhaustive selection** is inefficient
- **Random selection** may miss some values
- FailureDoc selects replacement candidates by:
 - mapping each value to an **abstract** domain using an *abstract object profile* representation
 - sample each abstract domain



The architecture of FailureDoc



Execution result observation

- FailureDoc executes each mutated test, and classifies it as:
 - **Passing**
 - **Failing**
 - The same failure as the original failed test
 - **Unexpected exception**
 - A different exception is thrown

Original test

```
...  
int i = 1;  
ArrayList lst = new ArrayList(i);  
...
```

Mutated test

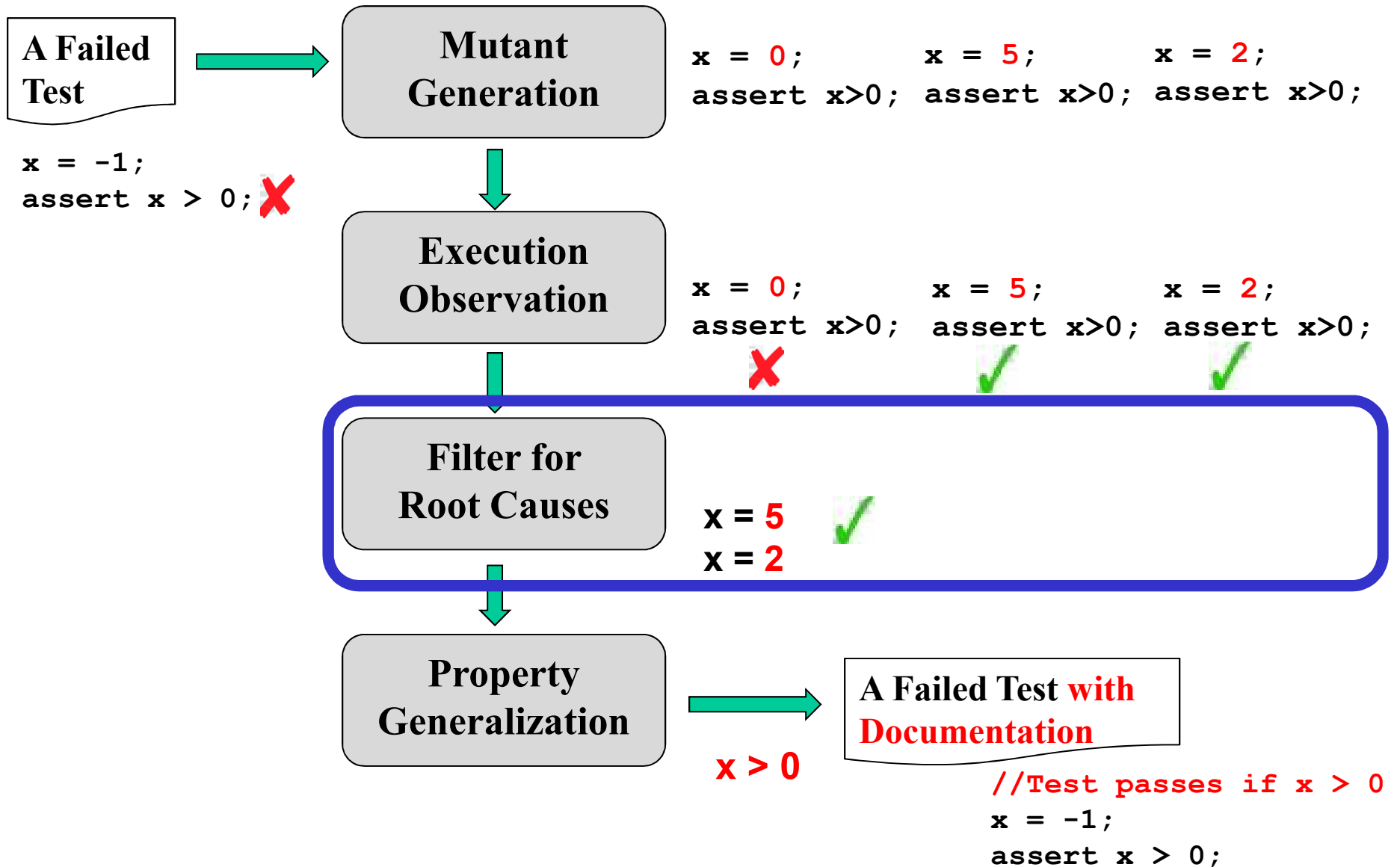
```
...  
int i = -10;  
ArrayList lst = new ArrayList(i);  
...
```

Unexpected exception: **IllegalArgumentException**

Record expression values in test execution

- After value replacement, FailureDoc only needs to record expressions that can affect the test result:
 - Computes **a backward static slice** from the assertion in passing and failing tests
 - **Selectively** records expression values in the slice

The architecture of FailureDoc



Statistical failure correlation

- A statistical algorithm isolates **suspicious statements** in a failed test
 - A variant of the CBI algorithms [Liblit'05]
 - Associate a suspicious statement with **a set of failure-correcting objects**
- Characterize the **likelihood** of each observed value v to be a failure-correcting object
 - Define 3 metrics: **Pass**, **Increase**, and **Importance** for each observed value v of each statement

Pass(v): the percentage of passing tests when **v** is observed

Original test	Observed value in a mutant
<pre>public void test1() { int i = 1; ArrayList lst = new ArrayList(i); Object o = new Object(); boolean b = lst.add(o); TreeSet ts = new TreeSet(lst); Set set = synchronizedSet(ts); //This assertion fails assertTrue(set.equals(set)); }</pre>	<p>b = false</p> <p>PASS!</p>

Pass(b=false) = 1

The test **always** passes, when **b** is observed as **false**

Pass(v): the percentage of passing tests when **v** is observed

Original test	Observed value in a mutant
<pre>public void test1() { int i = 1; ArrayList lst = new ArrayList(i); Object o = new Object(); boolean b = lst.add(o); TreeSet ts = new TreeSet(lst); Set set = synchronizedSet(ts); //This assertion fails assertTrue(set.equals(set)); }</pre>	<pre>ts = an empty set PASS!</pre>

Pass(*ts = an empty set*) = 1

The test **always** passes, when **ts** is observed as *an empty set*!

Increase(v): indicating root cause for test passing

Original test	Observed value in a mutant
<pre>public void test1() { int i = 1; ArrayList lst = new ArrayList(i); Object o = new Object(); boolean b = lst.add(o); TreeSet ts = new TreeSet(lst); Set set = synchronizedSet(ts); //This assertion fails assertTrue(set.equals(set)); }</pre>	<p>Changing b's initializer to false implies ts is an empty set</p> <p>b = false ts = an empty set</p> <p>PASS!</p>

Increase(b = false) = 1
Increase(ts = an empty set) = 0

Distinguish the *difference* each observed value makes

Importance (v) :

- harmonic mean of ***increase***(v) and the *ratio of passing tests*
- balance sensitivity and specificity
- prefer high score in both dimensions

Algorithm for isolating suspicious statements

Input: a failed test t

Output: suspicious statements with their *failure-correcting* objects

Statement s is suspicious if its *failure-correcting object set* $FC_s \neq \emptyset$

$FC_s = \{v \mid$

$Pass(v) = 1 \quad \wedge \quad /* v \text{ corrects the failed test } */$

$Increase(v) > 0 \quad \wedge \quad /* v \text{ is a root cause } */$

$Importance(v) > \text{threshold} \quad /* \text{balance sensitivity \& specificity } */$

$\}$

Failure-correcting objects for the example

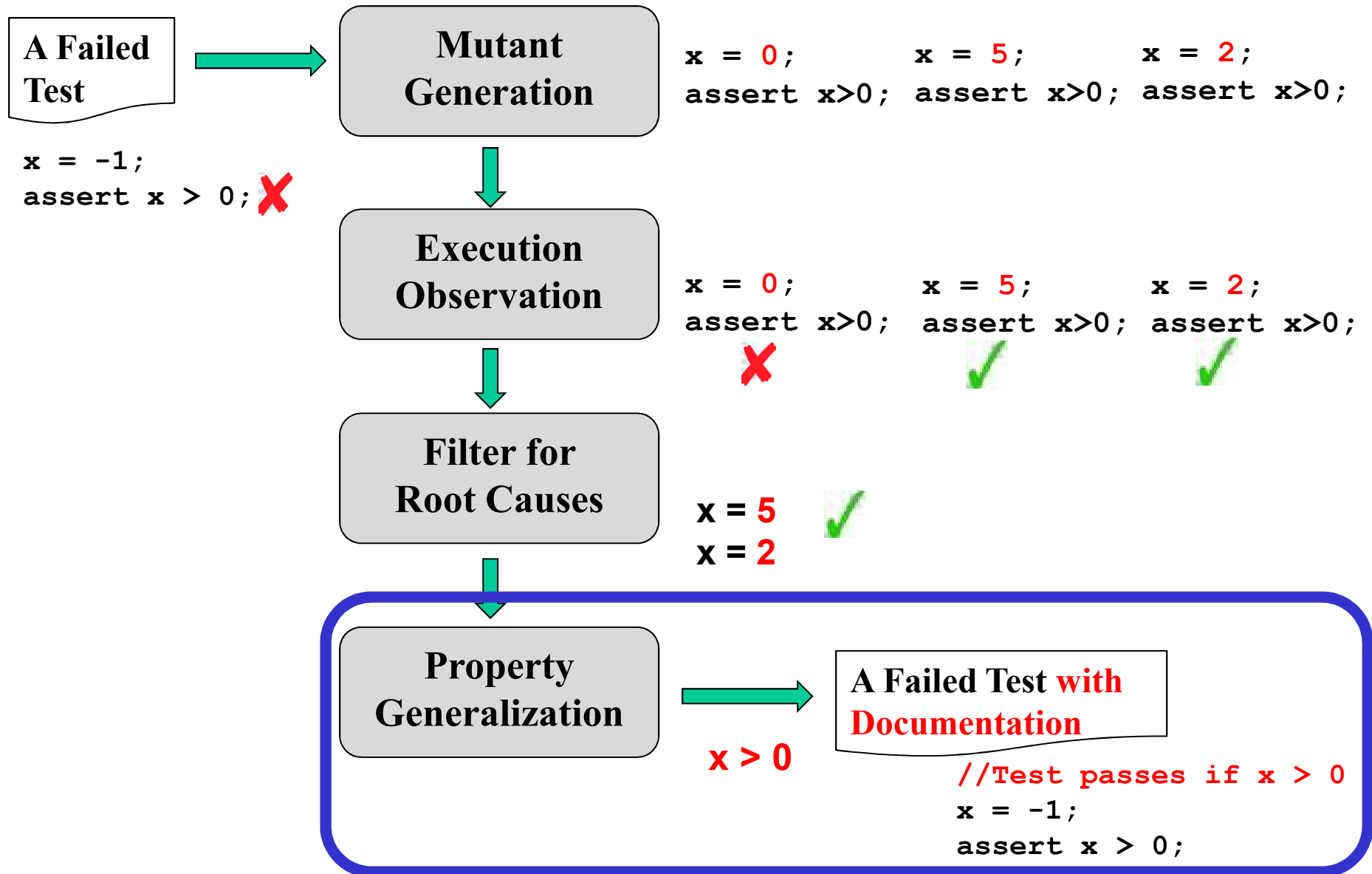
Original test

```
public void test1() {
    int i = 1;
    ArrayList lst = new ArrayList(i);
    Object o = new Object();
    boolean b = lst.add(o);
    TreeSet ts = new TreeSet(lst);
    Set set = synchronizedSet(ts);
    //This assertion fails
    assertTrue(set.equals(set));
}
```

Failure-correcting object set

```
o ∈ {100, (byte)1, "hi"}
b ∈ {false }
```


The architecture of FailureDoc



Property generalization

- Generalize properties for **failure-correcting objects**
 - Use a Daikon-like technique
 - E.g., property of the object set: {100, "hi!", (byte) 1} is:
all values are comparable.
- Rephrase properties into readable documentation
 - Employ a small set of templates:
x instanceof Comparable ⇒ **x implements Comparable**
x.add(y) replaced by false ⇒ **y is not added to x**

Outline

- Overview
- The **FailureDoc** technique
-  • Implementation & Evaluation
- Related work
- Conclusion

Research questions

- **RQ1:** can FailureDoc infer explanatory documentation for failed tests?
- **RQ2:** is the documentation useful for programmers to understand the test and fix the bug?

Evaluation procedure

- An experiment to explain **12 failed tests** from 5 subjects
 - All tests were automatically generated by Randoop [[Pacheco'07](#)]
 - Each test reveals a distinct real bug
- A user study to investigate the documentation's usefulness
 - 16 CS graduate students
 - Compare the **time cost** in **test understanding** and **bug fixing** :
 1. **Original tests (undocumented)** vs. **FailureDoc**
 2. **Delta debugging** vs. **FailureDoc**

Subjects used in explaining failed tests

Subject	Lines of Code	# Failed Tests	Test size
Time and Money	2,372	2	81
Commons Primitives	9,368	2	150
Commons Math	14,469	3	144
Commons Collections	55,400	3	83
java.util	48,026	2	27

- Average test size: **41** statements
- Almost all failed tests involve complex interactions between **multiple classes**
 - Hard to tell why they fail by simply looking at the test code

Results for explaining failed tests

- FailureDoc infers meaningful documentation for **10** out of 12 failed tests
 - Time cost is **acceptable**: **189 seconds** per test
 - Documentation is **concise**: **1 comment per 17 lines** of test code
 - Documentation is **accurate**: each comment indicates a different way to make the test pass, and is ***never in conflict*** with each other
- FailureDoc fails to infer documentation for **2** tests:
 - no way to use value replacement to correct them

Feedback from developers

- We sent all documented tests to subject developers, and got positive feedback
- Feedback from a Commons Math developer:

*I think these comments **are helpful**. They give a hint about what to look at. ... **the comment showed me exactly the variable to look at.***
- Documented tests and communications with developers are available at: <http://www.cs.washington.edu/homes/szhang/failedoc/bugreports/>

User study: how useful is the documentation?

- Participants: 16 graduate students majoring in CS
 - Java experience: max = 7, min = 1, avg = 4.1 **years**
 - JUnit experience: max = 4, min = 0.1, avg = 1.9 **years**
- 3 experimental treatments:
 - ***Original tests (undocumented)***
 - ***Delta-debugging-annotated tests***
 - ***FailureDoc-documented tests***
- Measure:
 - time to **understand why a test fails**
 - time to **fix the bug**
 - 30-min time limit per test

Results of comparing *undocumented tests* with *FailureDoc*

Goal	Success Rate		Average Time Used (min)	
	JUnit	FailureDoc	JUnit	FailureDoc
Understand Failure	75%	75%	22.6	19.9
Understand Failure + Fix Bug	35%	35%	27.5	26.9

JUnit: Undocumented Tests

FailureDoc: Tests with FailureDoc-inferred documentation

Conclusion:

- FailureDoc helps participants *understand a failed test* 2.7 mins (or 14%) *faster*
- FailureDoc *slightly speeds up* the bug fixing time (0.6 min faster)

Results of comparing *Delta debugging* with *FailureDoc*

Goal	Success Rate		Average Time Used (min)	
	DD	FailureDoc	DD	FailureDoc
Understand Failure	75%	75%	21.7	20.0
Understand Failure + Fix Bug	40%	45%	26.1	26.5

DD: Tests annotated with **Delta-Debugging**-isolated faulty statements
Delta debugging can only isolate faulty statements in 3 tests

FailureDoc: Tests with FailureDoc-inferred documentation

Conclusion:

- FailureDoc helps participants *fix more bugs*
- FailureDoc helps participants to *understand a failed test faster* (1.7 mins or 8.5%)
- Participants spent *slightly more time* (0.4 min) in fixing a bug on average with FailureDoc, though more bugs were fixed

Feedback from Participants

- Overall feedback

- FailureDoc is useful
- FailureDoc is *more useful* than Delta Debugging

- Positive feedback

The comment at line 68 did provide information *very close to* the bug!

The comments are useful, because they indicate *which variables are suspicious*, and help me *narrow the search space*.


- Negative feedback

The comments, though [they] give useful information, can *easily be misunderstood*, when I am *not familiar* with the [program].

Experiment discussion & conclusion

- Threats to validity
 - Have not used human-written tests yet.
 - Limited user study, small tasks, a small sample of people, and unfamiliar code (is 30 min per test enough?)
- Experiment conclusion
 - FailureDoc can infer *concise and meaningful* documentation
 - The inferred documentation is *useful* in understanding a failed test

Outline

- Overview
- The **FailureDoc** technique
- Implementation & Evaluation
-  • Related work
- Conclusion

Related work

- **Automated test generation**

Random [[Pacheco'07](#)], Exhaustive [[Marinov'03](#)], Systematic [[Sen'05](#)] ...

Generate new tests instead of explaining the existing tests

- **Fault localization**

Testing-based [[Jones'04](#)], delta debugging [[Zeller'99](#)], statistical [[Liblit'05](#)] ...


Localize the bug in the tested code, but doesn't explain why a test fails

- **Documentation inference**

Method summarization [[Sridhara'10](#)], Java exception [[Buse'08](#)],
software changes [[Kim'09](#), [Buse'10](#)], API cross reference [[Long'09](#)]

Not applicable to tests (e.g., different granularity and techniques)

Outline

- Overview
- The **FailureDoc** technique
- Implementation & Evaluation
- Related work
-  • Conclusion

Future Work

- FailureDoc proposes *a different abstraction* to help programmers *understand a failed test*, and *fix a bug*.

Is there a better way?

- Which information is *more useful* for programmers?
 - Fault localization: pinpointing the buggy program entities
 - Simplifying a failing test
 - Inferring explanatory documentation
 -

Need more experiments and studies

Contributions

- *FailureDoc*: an automated technique to explain failed tests
 - Mutant Generation
 - Execution Observation
 - Statistical Failure Correlation
 - Property Generalization

- An open-source tool implementation, available at:



<http://failuredoc.googlecode.com/>

- An experiment and a user study to show its usefulness
 - Also compared with Delta debugging

[Backup slides]

Comparison with Delta debugging

- ***Delta debugging:***
 - **Inputs:** A passing and a failing version of a program
 - **Output:** failure-inducing edits
 - **Methodology:** systematically explore the change space
- ***FailureDoc:***
 - **Inputs:** a single failing test
 - **Outputs:** high-level description to explain the test failure
 - **Methodology:** create a set of slightly-different tests, and generalize the failure-correcting edits

Comparison with the CBI algorithm

- The **CBI** algorithm:
 - **Goal:** identify likely buggy predicates in the *tested code*
 - **Input:** a large number of executions
 - **Method:** use the *boolean* value of an instrumented predicate as the feature vector
- Statistical failure correlation in **FailureDoc**
 - **Goal:** identify failure-relevant statements in *a test*
 - **Input:** a single failed execution
 - **Method:**
 - use *multiple observed values* to isolate suspicious statements.
 - associate each suspicious statement with a set of *failure-correcting objects*